THE UNIVERSITY *of* EDINBURGH

The experimental and theoretical validation of a new search algorithm, with a note on the automatic generation of causal explanations.

by

Kevin P. Coplan

Ph.D. University of Edinburgh 1983

ABSTRACT

An algorithm is presented for game-tree searching that is shown under fairly general but formally specifiable conditions to be more sparing of computational resource than classical alpha-beta minimax.

The algorithm was programmed in POP-2 and compared experimentally with alpha-beta searching on randomly generated trees, and the results are presented.

A machine for solving deep chess combinations was built from micro-electronic circuits. The general game-tree searching algorithm was embedded in the machine together with a chess-specific algorithm.

The chess-specific algorithm and the hardware of the machine are described.

The results of running the machine on selected chess positions are presented.

Deficiencies in the performance of the machine are described and improvements suggested.

The problem of generating human-oriented descriptions of combinatorial problems was considered using chess tactics as a domain.

A system is described for finding causal motivations for moves in a chess game-tree. The chess machine was

interfaced to a main-frame computer and programs were written which ran interactively with the chess machine to produce humanly understandable explanations of the combinations solved

The system was tested on selected positions and the results presented.

Deficiencies in the performance of the system are analysed and solutions suggested based on extensions of the underlying algorithm. Applicability of these methods is discussed to combinatorial problems encountered in industry and defence.

ACKNOWLEDGEMENTS

The work described in this thesis was done while the author was in receipt of an SRC Studentship in association with a project funded by SRC grant no. GR/A/80327 made to Professor Donald Michie for software and microelectronic aids for design and implementation of expert systems. The author acknowledges the University of Edinburgh's provision of facilities and expresses his thanks to Professor Donald Michie, Dr. Timothy Niblett and Dr. Ivan Bratko for their helpful suggestions and comments, also to Laura Mackay for her assistance in preparing the chess diagrams, and to Juliette Buss for her help during the construction phase of VIRGO.

TABLE OF CONTENTS

PLATES

Plate 1 is a front view of VIRGO.

Plate 2 shows the wiring sides of the logic boards.

Plate 3 shows the reverse side of the logic boards.

Notation and conventions

Diagram Convention.

In all the chess diagrams unless otherwise stated, White is to play. If in the source for the position, Black was to play, the position has been reflected about the horizontal line that bisects the board and the colours of the pieces reversed.

Node labelling.

Throughout the description of the search algorithm in chapter 2, each tree search is considered to be a game between "MAX" and "MIN". The root nodes of all trees are MAX nodes. Throughout the description of the VIRGO search machine and the explanation facility in chapters 3 and 4, MAX and MIN nodes are called UTM nodes and TTM nodes (for us to move and them to move) respectively.

Figure labelling.

In chapter 4 reference is made to certain chess positions that were used to test the explanation facility and which appear in appendix A. Where diagrams of these positions are also reproduced within the text of chapter 4, they have been labelled "POSITION n" where "n" is the number of the position in appendix A. All other figure numbering is

independent within each chapter.

POP-2 notation.

The POP-2 programming language is used in chapter 2 to describe the search algorithm. In particular, POP-2 expression lists are used in section 2.3. If a and b are expressions and a = 2 and b = 4, then [%a,b%] = [2,4].

CHAPTER 1

Introduction

For centuries the game of chess has challenged the highest levels of human intellect. Mastery of chess requires skills of planning, problem solving, reasoning, and learning by abstraction from examples. The advent of computers in recent years has presented a stronger challenge; that of programming a computer to play chess as well as any human. Although, since the 1950's much effort has been spent on this endeavour, even in 1983, no computer program plays chess to as high a standard as world champion chess players. It might be thought that the skills used by the best human players would be exhibited by the most successful chess playing programs. This is not the case, although attempts have been made. An example is a program by Pitrat [16] that solves chess combinations by executing plans: one of the problems with the program was the large number of plans that were generated as a result of combinatorial explosions. However in 1980 the World Computer Chess champion was the program "Belle" [8]. The advantage that enabled Belle to defeat its rival programs, was the speed at which it could search chess game-trees thanks to special hardware for tree searching which supported the program running on a DEC LSI-11 microcomputer.

World champion chess players and the best chess computer

programs use techniques which are markedly different from each other: humans using "intelligence" doing little search but much reasoning, and computer programs using "brute force", doing little reasoning but much search at relatively high speeds.

The greatest challenge in programming computers to play chess as well as the best human players, is not for the programs' chess playing ability in themselves, but for the discovery of principles of learning and reasoning that will have application in other fields of cognitive activity.

Chapters 2 and 3 of this thesis are concerned with "brute-force" techniques for computer chess playing. Chapter 2 describes a new algorithm for general game-tree searching, that is compared theoretically and experimentally with classical alpha-beta minimax. Chapter 3 describes a special-purpose machine built from low level micro-electronic components for solving tactical chess combinations. The algorithm described in chapter 2 is incorporated in the machine.

In chapter 4, the problem of generating human-oriented explanations to the combinations solved by the machine is investigated. A formalism is developed for describing basic tactical elements, understood by chess players as pins, forks, decoys, etc, in terms of the results of applying an algorithm for tracing reasons for moves, to

moves returned by the machine. Programs were written that implemented these ideas, and the results are presented.

It is intended that the methods used should help towards the development of chess playing programs with reasoning powers, and have application to combinatorial problems encountered in industry and defence.

CHAPTER 2.

Special search algorithm.

2.0 History of alpha-beta pruning.

The alpha-beta algorithm for pruning game-trees was known at least as early as 1956, when McCarthy thought of the method during the Dartmouth Summer Research Conference on Artificial Intelligence [10]. Knuth [10] gives a full description of the algorithm and proves two important results. Firstly that alpha-beta pruning is optimal in the sense that for any other algorithm that obtains the minimax value of a given game-tree, there is a re-ordering of the tree such that alpha-beta examines a subset of the terminal nodes examined by the other algorithm. Secondly that for a tree of uniform breadth b and depth d the smallest number of terminal nodes that will be examined is

```
        (2b ** (d/2)) -1                          for d even
and b ** (((d-1)/2) + b ** ((d+1)/2) -1)  for d odd
```

The performance of alpha-beta depends critically on the order in which successor nodes are generated, and various techniques have been developed for doing this ordering [2].

Knuth's theorem that alpha-beta is an "optimal" algorithm, depends on reordering the game-tree, as stated earlier. The SSS* algorithm [19] that makes use of a data structure of order b ** (d/2) elements, in performing a parallel minimax

search over a game-tree, has been shown to search a subset of the nodes examined by alpha-beta, if the tree is examined in the same order by both algorithms. This is a property shared by the game-tree search algorithm of this chapter.

2.1 Overview.

In this chapter an algorithm is presented for searching game trees, which is shown under certain circumstances to be more efficient than alpha-beta searching. The search method involves performing a number of searches, each with the task of deciding whether or not a particular goal value can be achieved. The actual minimax value of the tree is homed in on by binary chopping.

Section 2.2.1 describes the algorithm with an example. In section 2.2.2 the algorithm is presented formally in the form of a POP-2 program. In this section it is proved that the new-search method always examines a subset of the nodes that alpha-beta pruning examines and it is shown that this may be a proper subset.

In section 2.3.1 a method is described for storing subtrees of the tree being searched, and thereby reducing the number of nodes that need revisiting on subsequent searches.

In section 2.3.2 three methods are described and proved valid for pruning the stored subtrees, thus reducing the amount of storage required to contain these trees.

The algorithm was tested by generating a number of random trees and comparing the efficiency of the new-search method with alpha-beta pruning. In section 2.4 a description is given of the test runs and the results displayed.

2.2 Description and properties of the algorithm

2.2.1 Informal description by example.

Consider using the alpha-beta algorithm to search the tree in fig 1 below where values are maximised at MAX nodes and minimized at MIN nodes and the search takes place from left to right. Throughout this section, the root nodes of all trees are MAX nodes.

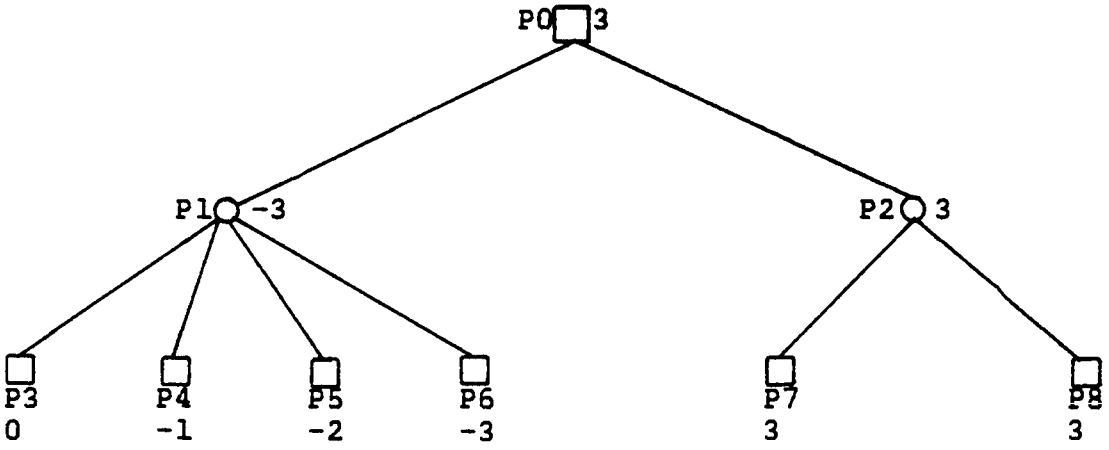


Fig 1

The first nodes to be searched are P3, P4, P5 and P6, which return the backed up value (BV) of -3 to the MIN node P1. P0 then gets the provisional backed up value (PBV) of -3. P7 and P8 are next looked at to give the min node P2 the BV of 3. Finally P0 gets the BV of 3. Note that no alpha-beta pruning has been possible. Suppose that instead of using

the alpha-beta algorithm, we do a series of searches, each search answering the question, "is it possible to obtain more than some value G at P0?" as well as backing up a numeric value to the root node.

Assume that we know that all the values at the terminal nodes will lie in the interval [-64,64]. Then set the initial value of G to 0. Assign to terminal nodes with numeric values $k>G$, the truth-value "true," and to terminal nodes with numeric values $k\leq G$ the truth-value "false," (see fig 2).

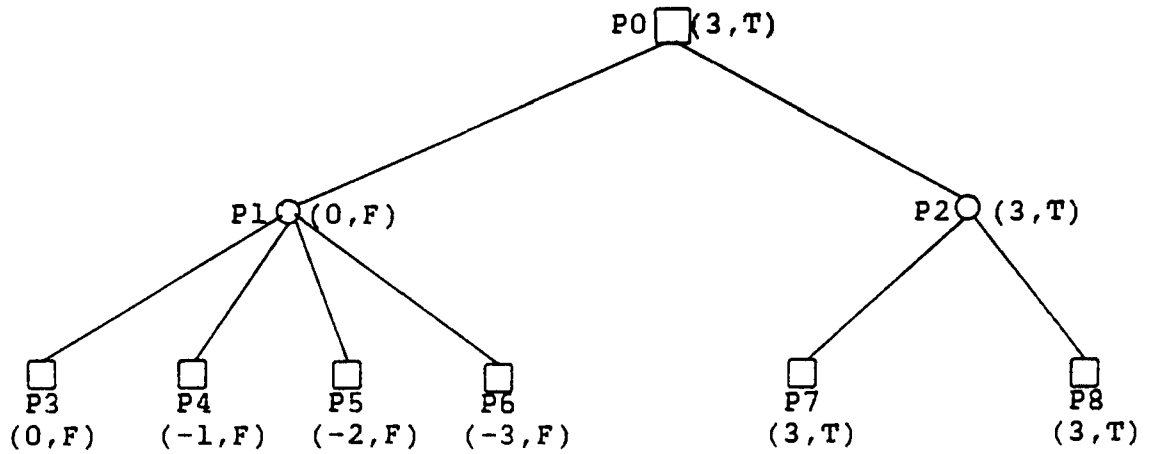


Fig 2. G=0.

Using the new type of search we first look at P3, and back up the pair of values (0,false) to P1. Since P1 is a MIN node and there is one branch from this node that returns the

value false, we do not look at P4, P5, P6. So the pair of values (0,false) is backed up to P0. We next look at P7 and P8, backing up the pair of values (3,true) to P2. Finally P0 gets the backed up pair of values (3,true).

We interpret this as meaning "there exists a strategy that yields us a score of at least 3." We now know that the value that would have been obtained by a minimax search lies in the interval [3,64], so we next set G to [(-64+3)/2] = 33. This gives us the tree shown in fig 3.

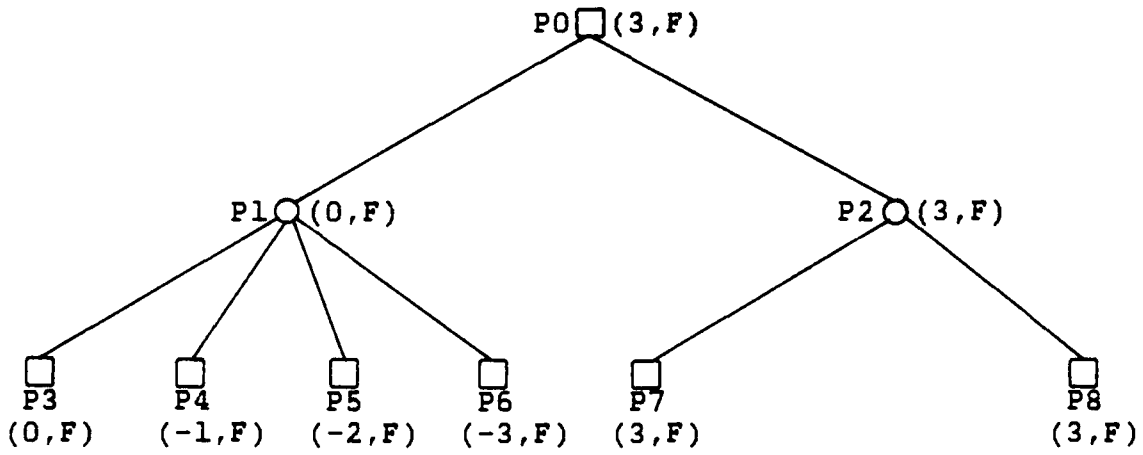


Fig 3. G=33.

We first look at P3 which gives us the pair of values (0,false). So (0,false) is backed up to P1. Since P1 is a MIN node and "false" has been backed up to it, we do not look at P4, P5, and P6, but return the values (0,false) to P0. We next look at P7 which returns (3,false) to P2.

Finally we ignore P8, and give P0 the values (3,false). We interpret this as meaning " there exists a strategy for "them" by which we can score at most 3." In other words the minimax BV would lie in the interval [-64,3]. But the tree in fig 1 showed us that it lay in [3,64]. So the minimax value must be 3.

Note that in neither of these searches have we looked at nodes P4, P5 and P6, but in the alpha-beta search we examined all nodes. In general we can prove that for all trees, all moves cut off by alpha-beta would also be cut off by this new type of search as long as moves are generated in the same order at corresponding nodes of both trees.

The above example shows that there is a tree with nodes cut off by new-type pruning, but not by alpha-beta. So new-type searching looks at a subset of the nodes alpha-beta looks at and sometimes a proper subset. However this does not necessarily imply that new-type searching is more efficient than alpha-beta searching, because the union of the separate trees produced by a new-type search is not a disjoint union. Thus the total number of nodes visited, including re-visited nodes may be greater than the number of nodes visited in the corresponding alpha-beta search.

2.2.2 The algorithm and its properties.

Let T be a tree with numeric values assigned to its terminal nodes. Let i be a numeric value. Then we can search a tree T with goal value i by the following POP-2 program. The program takes a position P as its argument and returns the pair of values BVN and BVT as its result. The tree newTi, is the subtree of T consisting of the nodes visited by the program.

```
function MAX P;
        if terminal(P) then
                value(P) -> BVN(P);
                value(P) > i -> BVT(P);
        else
                firstsuc(P) -> P+;
                MIN(P+);
                BVN(P+) -> PBV;
                if PBV > i then
                        PBV -> BVN(P);
                        true -> BVT(P);
                exit;
                until last(P,P+) then
                        next(P,P+) -> P+;
                        MIN(P+);
                        max(BVN(P+),PBV) -> PBV;
                        if PBV > i then
                                PBV -> BVN(P);
                                true -> BVT(P);
                        exit;
                close;
                PBV -> BVN(P);
                false -> BVT(P);
        close;
end;
```

```
function MIN P;
        if terminal(P) then
                value(P) -> BVN(P);
                value(P)>i -> BVT(P);

        else
                firstsuc(P) -> P+;
                MAX(P+);
                BVN(P+) -> PBV;
                if PBV ≤ i then
                        PBV->BVN(P);
                        false->BVT(P)
                exit;
                until last(P,P+) then
                        next(P,P+) -> P+;
                        MAX(P+);
                        min(BVN(P+),PBV) -> PBV;
                        if PBV ≤ i then
                                PBV->BVN(P);
                                false->BVT(P);
                        exit;
                close;
                PBV->BVN(P);
                true->BVT(P);
        close;
end;
```

Where P is a position,

terminal(P) is true iff P is a terminal node,

value(P) is the value assigned to terminal node P in tree T,

last(P,P+) is true iff P+ it the last successor node of P,

next(P,P+) is the next successor node of P after P+.

firstsuc(P) is the first successor of P.

i is a global variable set to the goal value.

P+ is used denote a successor node of P and

P- is used denote the parent node of P.

The backed up value, BV of a node P in newTi is an ordered pair, the first component a numeric value and the second a truth-value, denoted by BVN and BVT respectively,

If P is a terminal node of newTi then

$$BVT(P) = \text{true if } BVN(P) > i$$

$$= \text{false otherwise.}$$

If P has successor nodes Pi, $1 \leq i \leq r$,

generated in newTi then;

for P a MAX node;

$$BVN(P) = \max_i BVN(P)$$

$$BVT(P) = \bigcup_i BVT(P)$$

for P a MIN node;

$$BVN(P) = \min_i BVN(Pi)$$

$$BVT(P) = \bigcap_i BVT(Pi)$$

Cutoffs.

If P is a MAX node in newTi then a cutoff occurs as soon as one of the successor nodes of P returns a BV with truth component true. After that no more successor moves of P are examined. If P is a MIN node in newTi then a cutoff gives a BV with truth component false

Theorem.

Let T be a tree with numeric values assigned to its terminal nodes, and let abT and newTi denote respectively the subtrees of T derived by an alpha-beta search and a new type search with goal i.

i.e. T is the exhaustive minimax tree.

abT is the tree visited by an alpha-beta search on T.

newTi is the tree visited by a new-search on T with goal i.

Then newTi is a subtree of abT. It is assumed that moves are generated in the same order at corresponding nodes of abT and newTi.

Proof.

Throughout this proof backed up values at nodes in new type trees will refer only to their truth-value components.

Let s be a node in T which is not in abT. We wish to show that s is not in newTi. Since s has been cut off by alpha-beta pruning, s must be in scope of some position P1 in abT at which the cutoff took place. So there was a move M1; a successor move to P1, which returned a BV to P1 of a, say, and caused the cutoff. M1 was then the last successor move of P1 to be generated.

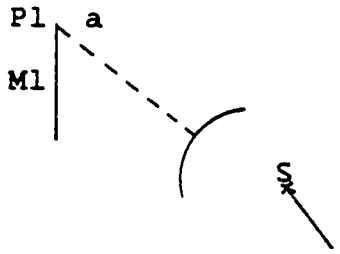
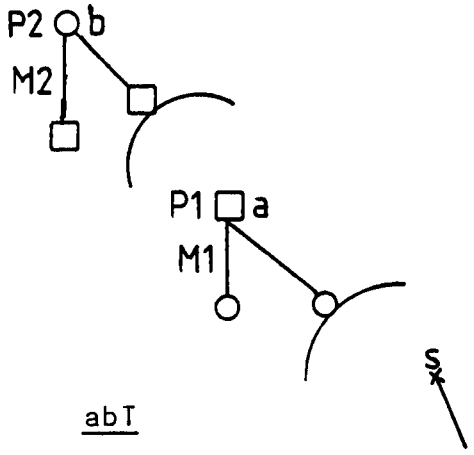


Fig 4.

Case 1. P1 is a MAX node.



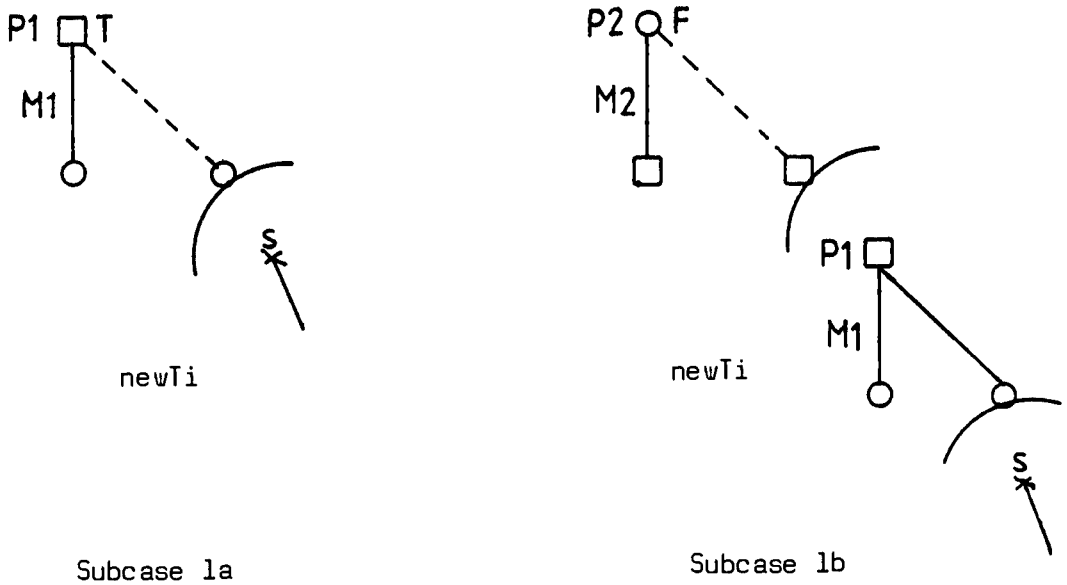
Case 1

Fig 5.

Then there exists a MIN node P2 which has P1 in its scope, where there is a PBV of b say at P2 such that a≥b.

Subcase 1a. i<a.

Then if P1 is in newTi M1 would have returned a BV of true to P1, by proposition 1 (proved later) so that there would be a cutoff at P1, after M1, and s would not be in newTi; as required. (See fig 6).



Subcase 1a

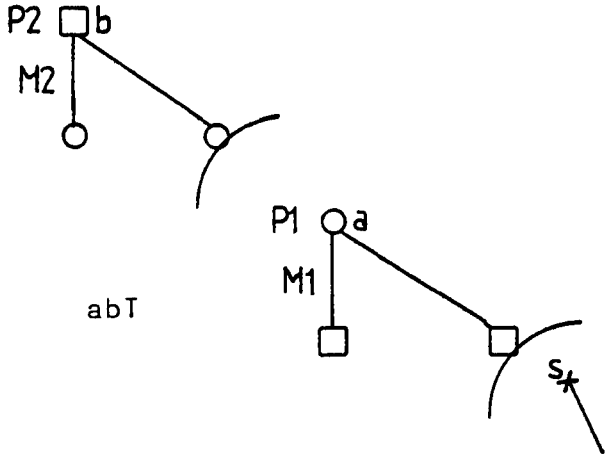
Subcase 1b

Fig 6.

Subcase 1b. $i \geq a$.

Let M2 be the move that gave P2 the PBV b. Then since $i \geq a$ and $a \geq b$, $i \geq b$. Therefore if P2 was generated in newTi, M2 would have returned the value false to P2 by proposition 1, stated and proven later. But then there would have been a cutoff at P2 after M2, and s would not be in newTi.

Case 2. P1 is a MIN node.



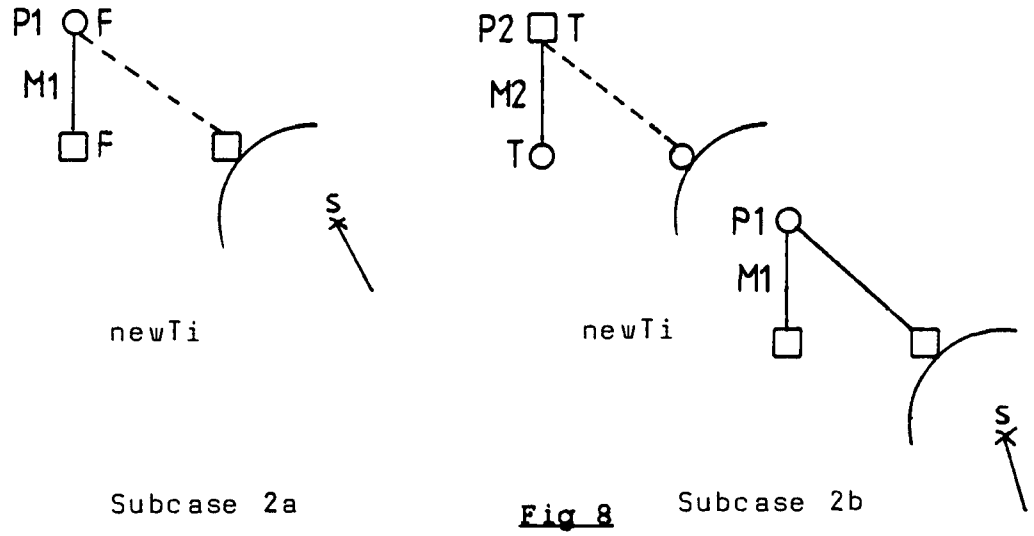
Case 2

Fig 7.

Then there exists a MAX node P2 which has P1 in its scope. There must be a PBV of b, say at P2 such that a≤b.

Subcase 2a. a≤i.

Then if P1 was in newTi, M1 would have returned a BV of false to P1 by proposition 1, so there would be a cutoff at P1 after M1, and so s is not in newTi; as required.



Subcase 2a

Fig 8

Subcase 2b

Subcase 2b. a>i.

Let M2 be the move that gave P2 the PBV b. Then since a>i and b≥a it follows that b>i. Therefore, if P2 was generated in newTi, M2 would have returned the value true to P2 by proposition 1. But then there would have been a cutoff at P2 after M2, and s would not be in newTi.

Proposition 1.

If P is a node which occurs in both abT and newTi, then BV(P) in abT is greater than i, iff BVT(P) in newTi is true.

Proof

Suppose not. Then in the alpha-beta search, there is a first node P to get a final backed up value which fails to satisfy the above property. There are then four cases;

Case (1) P is a MAX node and BV(P) > i

(2) P is a MAX node and BV(P) $\leq$ i

(3) P is a MIN node and BV(P) > i

(4) P is a MIN node and BV(P) $\leq$ i

Case (1). P is a MAX node and BV(P) > i.

Since P does not satisy the proposition, we must have BV(P) = false in newTi (see fig 9). Therefore all successor moves of P were generated in newTi and returned the value false. But then all the successor moves of P which were generated in abT had BV's < i since they received their BV's before P, and therefore satisfy the proposition.

Fig 9.

But then BV(P) $\leq$ i, which is a contradiction.

Case (2). P is a MAX node and BV(P) $\leq$ i.

Since P does not satisfy the proposition, we must have BV(P) = true in newTi.

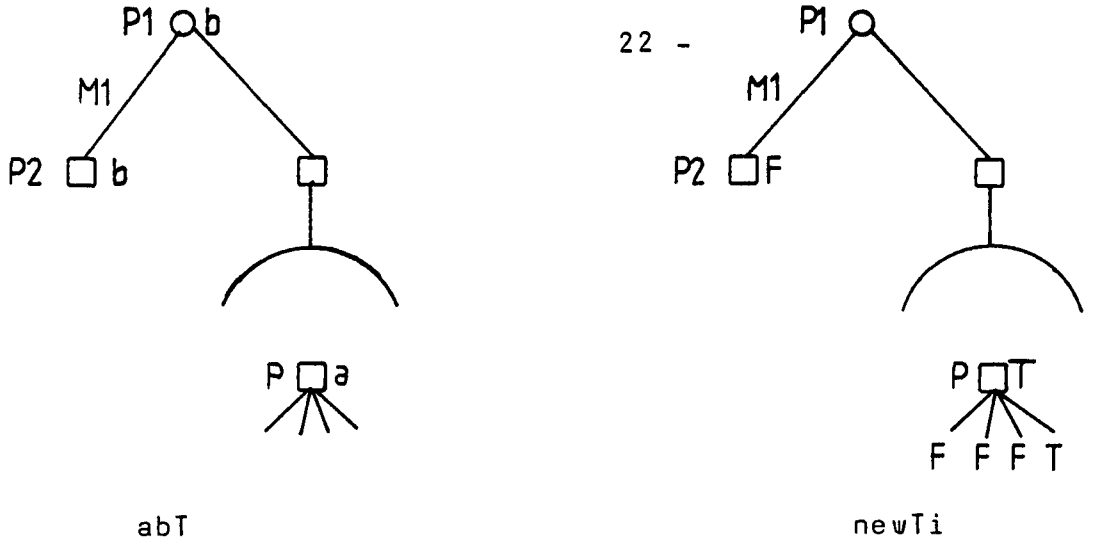


Fig 10.

Let BV(P) = a Then a ≤ i (see fig 10). Since BV(P) in newTi is true there must be a successor move of P in newTi which returned a BV of true to P. If this move was generated by the alpha-beta search in abT it would have returned a BV > i to P because it gets its BV before P does and hence satisfies the proposition. But then a>i, which is a contradiction, so there must have been an alpha-beta cutoff at P. therefore there is a node P1 in abT with a PBV b, s.t. b ≤ a. So there is a successor move M1 of P1 leading to the MAX position P2 which had a BV of b (see fig 10).

But P2 got its BV before P did, so since b ≤ a ≤ i, we can deduce that BV(P2) = false in newTi. But P1 is a MIN node and so a cutoff would have occurred at P1 immediately after M1 in newTi. So P is not in newTi. We therefore have a contradiction for case (2).

Cases (3) and (4) are proved similarly.

2.3 The use of memory for reducing search.

2.3.1 The function "mem".

In section 2.2 it was shown that the new search algorithm looks at a subset of the nodes that alpha-beta pruning looks at. However the new search algorithm is not necessarily more efficient than alpha-beta, as some nodes need to be revisited. The efficiency of the algorithm can be improved by storing backed up values obtained during each search and using them to effect further pruning in subsequent searches. At each visited node P, a pair of values [BVL,BVH] can be stored, where BVL and BVH are the least and greatest possible values for the minimax value of P. On any subsequent search with goal g, if P is revisited, a check can be made to see whether:-

(i) $g<BVL$

(ii) $g\geq BVH$

(iii) neither (i) nor (ii).

In case (i) the search backs up from P with BVT(P)=true and BVN(P)=BVL. In case (ii) the search backs up from P with BVT(P)=false and BVN(P)=BVH. In case (iii) the search continues from P.

Example

Consider the tree in fig 11 below, where the range of the

integers at the terminal nodes lie in [1,8].

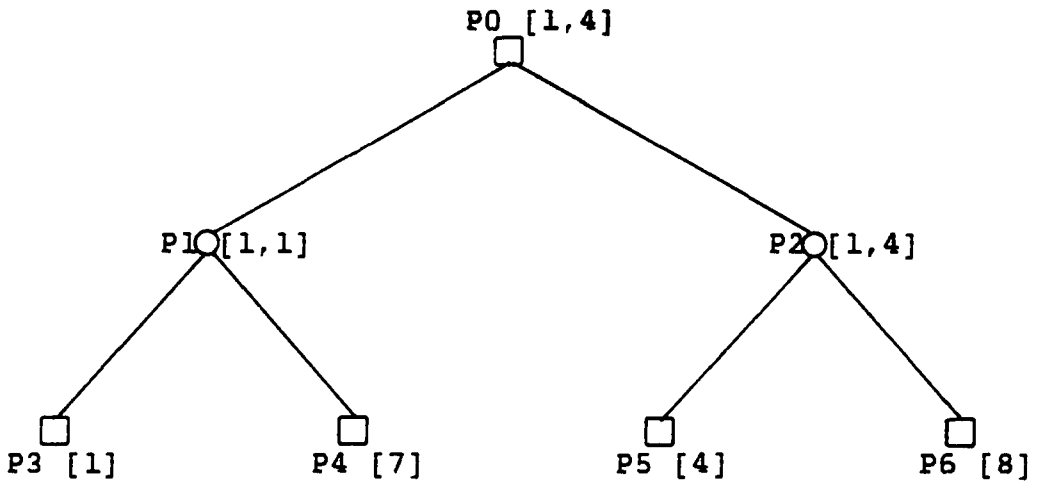


Fig 11.

If the goal=4 on the first search, then the first node examined is P3, which returns the BV 1 to P1 and causes P4 to be cutoff. We now know that the minimax value of P1 must be smaller than or equal to 1. Thus BVH at P1 is 1. Since 1 is the smallest value in the range, BVL=1 at P1. We therefore store the pair [1,1] at P1 (see fig 11).

After P5 has been visited [1,4] is stored at P2, and then [1,4] is stored at P0. The search has failed, so the next goal value is [(1+4)/2]. As soon as P1 is entered, a check is made to see which of the three conditions above, the value g satisfies. It is found that case(ii) holds, so that instead of revisiting P3, the search backs up from P1 to P0.

If BVL(P)i and BVH(P)i represent lower and upper bounds for the minimax value of a node P on the ith search, then BVL

and BVH are given by;

for P a MAX non-terminal node;

BVL(P)i=max{BVL(P)i-1,max{BVL(P+)i |
for all successors P+ of P
visited on the ith search}}

BVL(P)0 =bottom.

BVH(P)i=min {BVH(P)i-1,max{BVH(P+)i |
for all successors P+ of P}}
if all successors of P were
visited on the ith search.

=BVH(P)i-1 otherwise.

BVH(P)0=top.

Top and bottom represent respectively the greatest and least values in the range.

for P a MIN non-terminal node;

BVL(P)i=max{BVL(P)i-1,min{BVL(P+)i |

for all successors P+ of P}}

if all successors of P were

visited on the ith search.

=BVL(P)i-1 otherwise.

BVL(P)0=bottom

BVH(P)i=min{BVH(P)i-1,min{BVH(P+)i |

for all successors P+ of P

generated on the ith search}}.

BVH(P)0=top.

for P a terminal node;

BVL(P)i = BVH(P)i = BVN(P)

With these definitions it can be shown that:-

$$BVL(P)i \preceq minimax(P) \preceq BVH(P)i$$

We can now define the function "mem" which stores the backed up values of nodes that have been generated. POP-2 expression lists are used in this definition (see Notation and conventions).

For P not a terminal node;

mem(P)=[%BVL(P),BVH(P)%]<>[%mem(P+),
listed in the order that the
P+'s were generated%]

For P a terminal node;

mem(P)=[%v%] where v is the value
at the terminal node.

For example in fig 11

mem(P0) = [%BVL(P0),BVH(P0)%] <> [%mem(P1),mem(P2)%]

After the search with goal=4 we have

BVL(P0)=1

BVH(P0)=4

mem(P1)=[1 1 [1] [7]]

mem(P2)=[4 4 [4] [8]]

so mem(P0)=[1 4 [1 1 [1][7]] [4 4 [4] [8]]]

2.3.2 Methods for pruning storage.

In practice, storing large trees can be expensive and can reduce the efficiency of the search since more frequent garbage collection calls may need to be made than if the assigned storage was free. It is therefore an advantage to continually prune the trees being stored.

1st storage reduction method

Let P be a MAX node with successors P1,P2,...Pn and suppose that on the ith search P is expanded and node Pr returns "true" ($1 \leq r \leq n$). Then the memory attached to all nodes P1,...Pr-1 can be freed since nodes P1,...Pr-1 need never be revisited. At the same time the value of r may be stored at P by the memory function, so that on subsequent searches the rth node is automatically generated first (see fig 12).

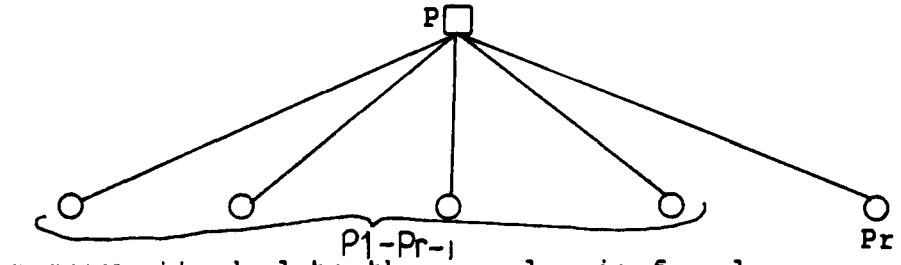


Fig 12.

Proof.

Let gi, gj be the goals on the ith and jth search

respectively where j>i. We then have two cases.

a) $gj \succeq gi$

b) $gj \prec gi$.

a) $gj \succeq gi$

Now it can be shown that a node P returns F on the ith search iff $BVH(P)i \preceq gi$ and P returns T on the ith search iff $BVL(P)i > gi$.

Since Pr was visited on the ith search, P1,...Pr-1 must all have returned false on the ith search.

$\Rightarrow BVH(Ps) \preceq gi \quad 1 \preceq s \preceq r-1$

But $gj \succeq gi$

$\Rightarrow BVH(Ps)i \preceq gj$.

From the definition of BVH(Ps) and the fact that $i \preceq j-1$,

$BVH(Ps)j-1 \preceq BVH(Ps)i$

$\Rightarrow BVH(Ps)j-1 \preceq gj$

Consequently, condition (ii) of section 2.3.1 would be satisfied for P1,...Pr-1 which would all return "false" without being expanded.

b) $gj \prec gi$.

From the definition of BVL(P)i, it follows that

$BVL(Pr)i \preceq BVL(P)i$

Now since Pr returned true on the ith search (as it caused a cutoff)

BVL(Pr)i $>$ gi

and since gi $>$ gj, we have

BVL(P)i $>$ gj.

From the definition of BVL(P)i and the fact that i$\leq$j-1, we have

BVL(P)j-1 $\geq$ BVL(P)i

$\Rightarrow$BVL(P)j-1 $>$ gj.

So by condition (i) of section 2.3.1 P would be exited with true, and P1,...Pr-1 would not be visited.

The same applies to moves causing cutoffs at MIN nodes.

2nd storage reduction method.

Values A and B corresponding to alpha and beta in an alpha-beta search, can be used to further prune the tree storage.

The values of A and B at the root node P0 are initialised to -infinity and +infinity. On descending on a node P+ from a parent node P;

A(P+) is initialised to A(P)

B(P+) is initialised to B(P)

A is updated at MAX nodes and B at MIN nodes by;

(i) If P is a MAX node and P+ a successor of P then in backing up from P+ to P;

A(P) is set to max{A(P),BVL(P+)}

(ii) If P is a MIN node and P+ a successor of P, then in backing up P+ to P;

B(P) is set to min{B(P),BVH(P+)}

On subsequent searches from P0, A(P0) and B(P0) are not reset to -infinity and +infinity, but keep their current values.

Lemma 1.

On any search with goal gi, in descending from a node P to a successor node Pr, the values of A and B at P satisfy

$A(P) \leq gi < B(P)$.

Proof.

Suppose move m between nodes P and Pr is the first move not to satisfy the above condition. Say P is a MAX node. There are two cases;

a) $A(P) > gi$;

b) $B(P) \leq gi$;

case a) A(P) > qi.

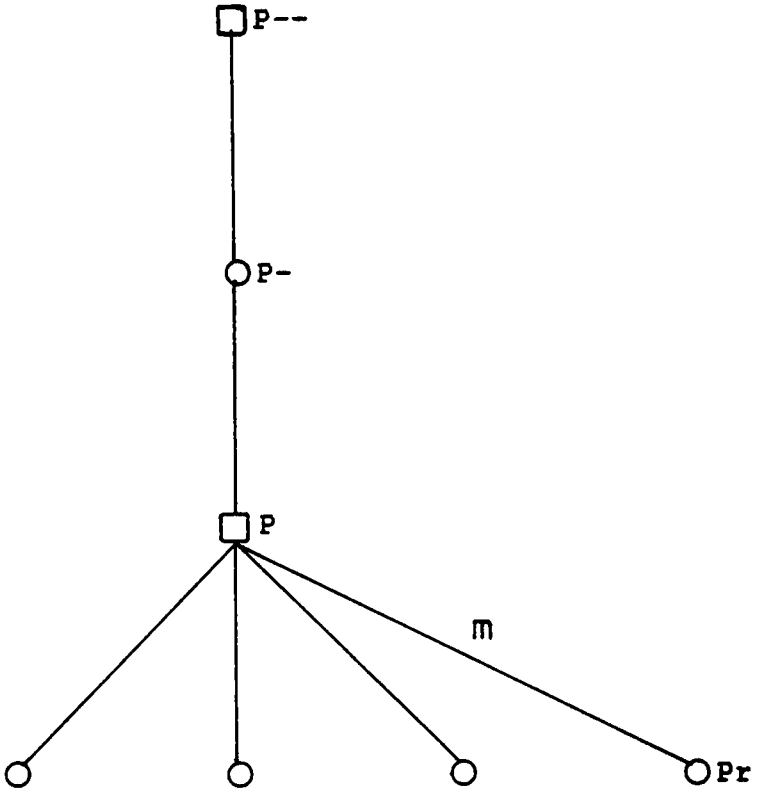


Fig 13.

If P is not the root node then after completing the search at node P

$$A(P)=\max\{A(P-),\max\{BVL(Ps) \mid 1\leq s\leq r-1\}\}$$

Since A is not updated at min nodes

$$A(P--) = A(P-)$$

$$\Rightarrow A(P) = \max\{A(P--),\max\{BVL(Ps) \mid 1\leq s \leq r-1 \}\}$$

If P0 is the root node, and A(P0)i-1 is the value of A at P0 after the i-1th search, then

$$A(P0)i = \max\{A(P0)i-1, \max\{BVL(Ps) \mid 1 \leq s \leq r-1\}\}.$$

There are then three possibilities;

(i) A(P) = A(P--)

(ii) A(P) = BVL(Ps) for some s: $1 \leq s \leq r-1$

(iii) A(P) = A(P0)i-1 if P is the root node P0.

(i) A(P) = A(P--)

Then on descent from P-- to P-, A(P--)>g. This contradicts P being the first node to have the property A(P) > gi or B(P) $\leq$ gi.

(ii) A(P) = BVL(Ps) for some s: $1 \leq s \leq r-1$.

BVL(Ps) < g forall s: $1 \leq s \leq r-1$, because each Ps returned false on the ith search. Otherwise a cutoff would have occurred before Pr was generated.

=> A(P) < g

Contradiction.

(iii) A(P) = A(P0)i-1 (P is the root node P0)

Then

I) A(P) = BVL(P+)j for some successor node P+ of P, $j \leq i-1$.

But $BVL(P+)j \leq BVL(P0)i-1$

and $BVL(P0)i-1 \leq gi$ by the way in which new goals are chosen.

$\Rightarrow A(P) \leq gi$.

contradiction.

or II) $A(P) \leq -infinity$

$\Rightarrow A(P) \leq gi$

case b) $B(P) \leq gi$.

P cannot be the root node because B(P0) = infinity always. So P has a parent node P-. Since the last time B(P) was updated (before the descent P -> Pr) was on the descent P- -> P, it follows that B(P-) = B(P). But then on descent from P- to P, $B(P-) \leq gi$. This contradicts move m being the first move to falsify the lemma.

The case where P is a MIN node is proved similarly.

Lemma 2

If Pr is a node that is visited on the ith and jth search where i<j, and Pr is the rth successor of P and m the move P->Pr then on descent from P to Pr;

$$A(P)i,m \leq A(P)j,m$$

$$B(P)i,m \geq B(P)j,m$$

where A(P)i,m represents the value of A at P on the ith search on descent through move m.

Proof.

Let move m between P and Pr be the first move to falsify the above condition. Say P is a MAX node.

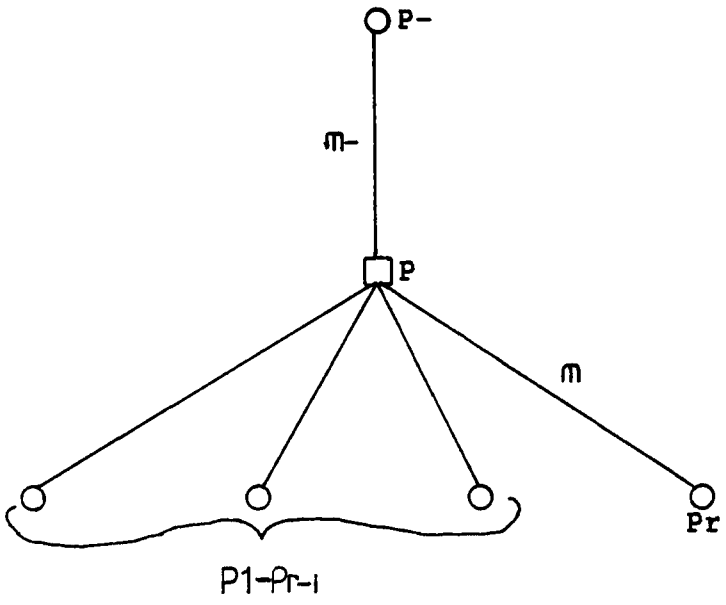


Fig 14.

Then either

a) $A(P)i,m > A(P)j,m$

or b) $B(P)i,m < B(P)j,m$

case a) $A(P)i,m > A(P)j,m$

$$A(P)j,m = \max\{A(P-)j,m\ ,\ \max\{BVL(Ps)j \mid 1 \leq s \leq r-1\}\}$$

$$A(P)i,m = \max\{A(P-)i,m\ ,\ \max\{BVL(Ps)i \mid 1 \leq s \leq r-1\}\}.$$

From the expression for A(P)i,m above it follows that either

I) $A(P)i,m = BVL(Ps)i$ for some s: $1 \leq s \leq r-1$

or II) $A(P)i,m = A(P-)i,m-$

If (I) $A(P)i,m = BVL(Ps)i$ for some $s: 1 \leq s \leq r-1$ then

$$BVL(Ps)i \leq BVL(Ps)j \leq A(P)j,m$$

$$\Rightarrow A(P)i,m \leq A(P)j,m$$

contradiction.

If (II) $A(P)i,m = A(P-)i,m-$

then $A(P-)i,m- \leq A(P-)j,m-$ since otherwise m- falsifies the lemma, contradicting m being the first such move.

so we have;

$$A(P)i,m = A(P-)i,m- \leq A(P-)j,m- \leq A(P)j,m$$

contradiction.

case b) $B(P)i,m < B(P)j,m$

Since B is only updated at MIN nodes and P is a MAX node

$$B(P)i,m = B(P-)i,m-$$

$$B(P)j,m = B(P-)j,m-$$

But then m- falsifies the lemma which contradicts m being the first such move.

If P is the root node P0, A(P) is only updated on backing up, from successor nodes P+, to max{A(P), BVL(P+)}. A(P0) is

not reset on successive searches, so in this case A(P)i < A(P)j whenever i<j. B(P0) remains constant to +infinity since P0 is not a MIN node. Thus the lemma holds for P = P0. If P is a MIN node, the proof is similar.

Proposition 2.

If Pr is the rth successor of P; m the move P->Pr, and

$$BVL(Pr)i > B(P)i,m$$

or $$BVH(Pr)i \leq A(P)i,m$$

then if Pr is visited on the jth search i<j, one of conditions (i) or (ii) of section 2.3.1 will be satisfied, and Pr will not be expanded.

Proof.

$$BVL(Pr)i > B(P)i,m$$

$$\Rightarrow \quad BVL(Pr)j-1 > B(P)i,m$$

(since $BVL(Pr)i \leq BVL(Pr)j-1$)

$\Rightarrow$ $BVL(Pr)_{j-1} > B(P)_{j,m}$

(since $B(P)_{j,m} \leq B(P)_{i,m}$ by lemma 2).

$\Rightarrow$ $BVL(Pr)_{j-1} > g_j$

(since $B(P)_{j,m} > g_j$ by lemma 1).

So that Pr satisfies condition (i) of section 2.3.1.

$BVH(Pr) \leq A(P)_{i,m}$

$\Rightarrow$ $BVH(Pr)_{j-1} \leq A(P)_{i,m}$

(since $BVH(Pr)_{j-1} \leq BVH(Pr)_i$)

$\Rightarrow$ $BVH(Pr)_{j-1} \leq A(P)_{j,m}$

(since $A(P)_{i,m} \leq A(P)_{j,m}$ by lemma 2).

$\Rightarrow$ $BVH(Pr)_{j-1} \leq g_j$

(since $A(P)_{j,m} \leq g_j$ by lemma 1).

so that Pr satisfies condition (ii) of section 2.3.1.

If the node Pr of proposition 2. is visited on the ith search and satisfies the given conditions, since Pr will not

be expanded on any subsequent searches, any memory assigned to the descendant nodes of Pr may be freed before backing up from Pr to P.

3rd storage reduction method.

If P+ is a successor node of P and P+ is visited on the ith search, and BVL(P)i = BVH(P)i then before backing up from P+ to P the memory assigned to the descendants of P+ can be freed since P+ will not be expanded on any subsequent searches. This is true because for i<j

either $BVL(P)i > gj$

or $BVH(P)i \leq gj$

But $BVL(P)i = BVL(P)j-1$

and $BVH(P)i = BVH(P)j-1$

since BVL(P) and BVH(P) cannot change after the ith search.

$\Rightarrow$ $BVL(P)j-1 > gj$

or $BVH(P)j-1 \leq gj$

So P will satisfy one of conditions (i) or (ii) of section 2.3.1 on descent from P- to P on the jth search.

2.4 Experimental testing of the algorithm.

2.4.1 Description of tests.

The new-search algorithm was compared experimentally with alpha-beta by generating trees of uniform breadth and fixed depth with random integers assigned to the terminal nodes.

For given breadths and depths, fifty random trees were generated, and the following quantities calculated;

N_{AB} The average number of nodes searched with alpha-beta pruning.

N The average number of nodes searched by the new-search method, including nodes not expanded due to conditions (i) or (ii) of section 2.3.1 being satisfied. Nodes visited n times are counted n times.

M The average number of nodes that were not expanded due to conditions (i) or (ii) of section 2.3.1 being satisfied.

N_D The average number of distinct nodes examined by the new-search method.

M_L The largest number of nodes stored in memory after each search. The size was calculated after each search with a given goal.

S The average number of repeat searches for each tree.

alb_bf This is $breadth^{[depth/2]}$ which is the absolute lowest bound of the branching factor on any single new-type search. (Only evaluated for depth = 4).

Only those trees, for which the full minimax tree contained less than 16,000 terminal nodes were searched. i.e. for which breadth ** depth < 16,000. The range of integers assigned to terminal nodes in each test was 0 to 63. The results are shown in tables 1-10 for depth=2 up to depth=13, and displayed on graphs 1-4 for depth=2 to depth=5.

Varying the range.

In each of the above tests the range of integers assigned to terminal nodes was fixed to [0,63]. A further run was made on trees of depth=6 and breadth=5, where the range was varied from [0,3] to [0,255] in powers of 2. The results are shown in table 9.

Normal distribution.

In all the above tests the distribution of random numbers was uniform, but in practice the distribution is more likely to be normal. Runs were made for trees of depth=6, breadth=5, range=256, where the distribution had mean 128 and standard deviations; 2, 4, 12, 16, 20, and 24. The results are shown in table 10.

2.4.2 Results.

breadth	N_{AB}	N/N_{AB} *100%	$(N-M)/N_{AB}$ *100%	N_D/N_{AB} *100%	M_L/N_{AB} *100%	S
2	6	132.4	116.7	100.0	83.3	1
12	80	150.5	108.4	91.3	30.0	4
22	253	136.4	109.6	95.7	17.8	5
32	427	87.6	67.6	59.0	15.2	5
42	615	118.0	94.1	84.2	13.7	6
52	945	112.3	91.5	82.9	11.0	6
62	1098	111.5	91.8	83.3	11.3	6
72	1256	112.8	92.3	83.8	11.5	6
82	1657	117.3	100.5	92.5	10.0	5
92	1913	118.4	102.1	94.1	9.6	5
102	2221	117.3	101.7	94.0	9.2	5
112	2678	116.7	101.3	94.1	8.4	5
122	3171	117.2	101.8	94.4	7.6	6

Table1. Depth=2.

The breadth in table 1 is incremented in steps of 10.

N_{AB}, N, M, N_D, M_L and S are as earlier defined in section 2.4.1.

breadth	N_{AB}	N/N_{AB} *100%	$(N-M)/N_{AB}$ *100%	N_D/N_{AB} *100%	M_L/N_{AB} *100%	S
2	13	132.5	117.9	92.3	46.2	2
3	30	122.1	109.9	93.3	26.7	3
4	52	114.0	103.0	88.5	19.2	3
5	83	110.6	101.0	90.4	14.5	3
6	121	100.3	92.7	85.1	11.6	3
7	186	95.5	87.8	81.2	8.6	4
8	241	99.7	91.0	84.6	7.5	4
9	320	94.9	87.4	83.1	6.3	4
10	450	84.6	80.5	77.6	4.9	4
11	541	84.7	81.1	78.4	4.4	4
12	578	89.7	85.1	82.4	4.5	5
13	773	79.0	76.1	74.3	3.6	4
14	963	80.8	78.4	76.7	3.1	4
15	862	99.5	93.1	89.6	3.7	5
16	933	91.8	87.1	84.4	3.6	4
17	1336	70.9	67.3	65.1	2.7	5
18	1117	94.8	88.5	85.3	3.4	5
19	1129	99.0	92.3	88.8	3.5	5
20	1307	99.7	93.5	90.0	3.2	5
21	1767	99.3	94.4	91.5	2.5	6
22	1979	98.0	93.4	90.8	2.3	6
23	2047	98.1	93.0	90.5	2.3	5
24	2584	96.9	92.4	90.3	1.9	5
25	3092	96.6	92.7	90.7	1.7	6

Table 2. Depth=3.

N_{AB}, N, M, N_D, M_L and S are as earlier defined in section 2.4.1.

breadth	N_{AB}	N/N_{AB} *100%	$(N-M)/N_{AB}$ *100%	N_D/N_{AB} *100%	M_L/N_{AB} *100%	S	alb_bf
2	25	116.0	105.5	88.0	52.0	2	4
3	68	110.6	100.5	85.3	36.8	3	9
4	166	94.0	87.2	79.5	24.1	3	16
5	311	89.3	80.7	73.0	19.3	3	25
6	621	101.0	91.6	83.4	13.7	5	36
7	990	99.0	89.9	83.2	11.3	5	49
8	1003	93.6	84.5	77.9	14.4	5	64
9	2249	79.7	74.3	70.5	8.0	6	81
10	2473	91.1	83.6	79.0	8.9	6	100
11	2847	97.4	89.1	84.1	9.3	5	121

Table 3. Depth=4.

breadth	N_{AB}	N/N_{AB} *100%	$(N-M)/N_{AB}$ *100%	N_D/N_{AB} *100%	M_L/N_{AB} *100%	S
2	45	115.7	107.1	88.9	31.1	2
3	158	103.9	95.5	85.4	16.5	4
4	504	86.9	83.3	78.2	8.3	4
5	1116	78.7	75.1	71.2	5.6	5
6	2450	78.9	76.3	73.8	3.5	5

Table 4. Depth=5.

N_{AB}, N, M, N_D, M_L and S are as earlier defined in section 2.4.1.

breadth	N_{AB}	N/N_{AB} *100%	$(N-M)/N_{AB}$ *100%	N_D/N_{AB} *100%	M_L/N_{AB} *100%	S
2	80	106.3	99.7	86.3	33.7	2
3	470	96.9	91.6	83.8	16.4	4
4	990	93.1	87.0	80.1	16.6	4
5	3283	81.0	77.1	73.1	9.4	5

Table 5. Depth=6.

breadth	N_{AB}	N/N_{AB} *100%	$(N-M)/N_{AB}$ *100%	N_D/N_{AB} *100%	M_L/N_{AB} *100%	S
2	143	98.0	93.1	81.1	20.3	3
3	955	88.5	85.0	79.5	8.4	5

Table 6. Depth=7.

N_{AB}, N, M, N_D, M_L and S are as earlier defined in section 2.4.1.

breadth	N_{AB}	N/N_{AB} *100%	$(N-M)/N_{AB}$ *100%	N_D/N_{AB} *100%	M_L/N_{AB} *100%	S
2	265	95.4	91.2	80.8	21.9	4
3	2118	89.1	85.7	80.2	10.7	5

Table 7. Depth=8.

depth	N_{AB}	N/N_{AB} *100%	$(N-M)/N_{AB}$ *100%	N_D/N_{AB} *100%	M_L/N_{AB} *100%	S
9	427	93.1	89.6	80.6	13.8	4
10	744	92.0	89.0	81.2	14.8	5
11	1231	88.0	85.7	79.2	9.0	5
12	2059	85.2	83.3	78.1	10.1	5
13	3359	84.9	83.3	78.6	6.2	5

Table 8. Breadth=2.

In table 8 the depth is varied in steps of 1, from 9 to 13.

N_{AB}, N, M, N_D, M_L and S are as earlier defined in section 2.4.1.

range	N_{AB}	N/N_{AB} *100%	$(N-M)/N_{AB}$ *100%	N_D/N_{AB} *100%	M_L/N_{AB} *100%	S
256	4639	69.4	67.1	64.4	6.6	6
128	3664	79.5	76.4	72.9	8.4	5
64	3283	81.0	77.1	73.1	9.4	5
32	3518	81.1	77.6	73.9	8.3	5
16	2637	83.7	81.0	77.7	10.7	4
8	2726	89.5	88.1	86.0	9.6	3
4	2179	105.2	102.7	99.2	9.5	2

Table 9. Breadth=5, Depth=6.

In table 9 the range is increased in powers of 2, from 4 to 256.

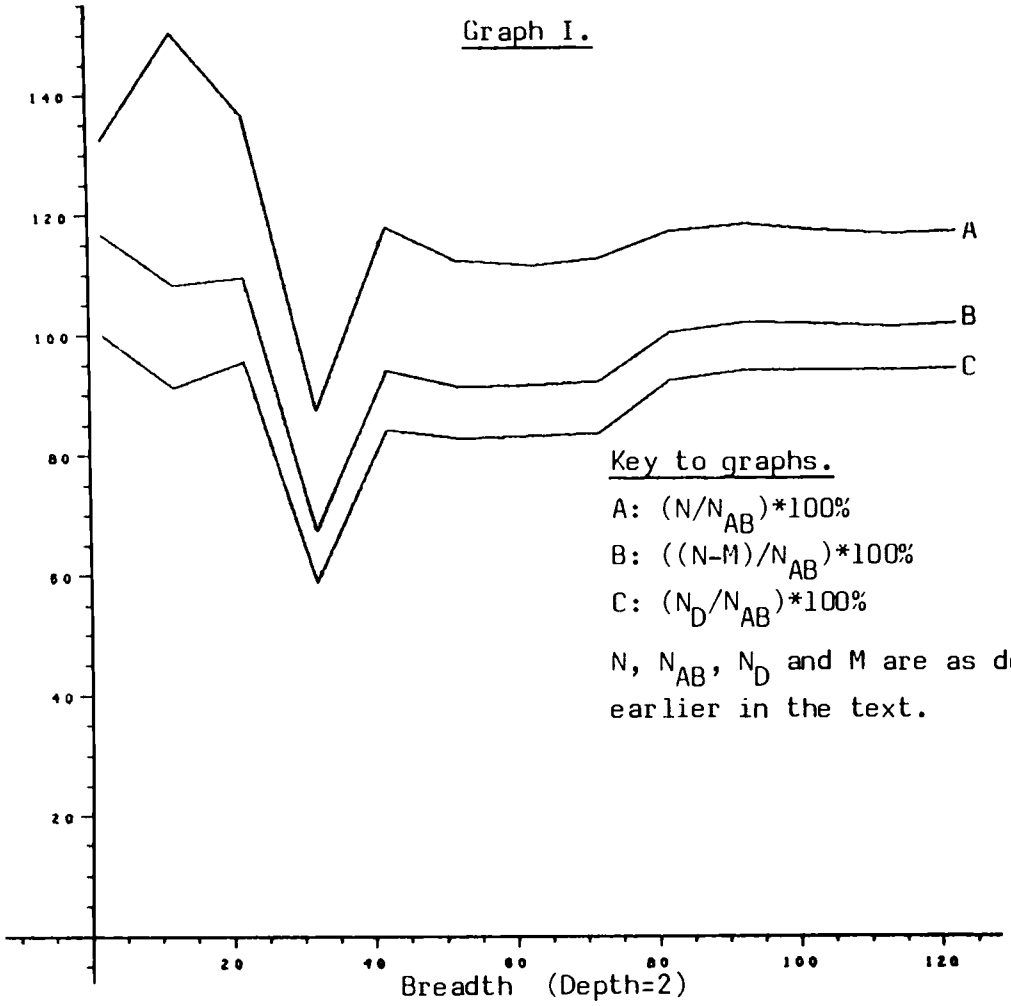
Standard deviation	N_{AB}	N/N_{AB} *100%	$(N-M)/N_{AB}$ *100%	N_D/N_{AB} *100%	M_L/N_{AB} *100%	S
24	3507	89.9	85.3	80.6	8.7	7
20	3820	95.7	91.0	86.3	7.7	8
16	3549	92.8	88.5	83.9	8.6	7
12	2528	100.9	93.9	87.2	12.1	6
8	2654	98.3	93.7	88.4	11.5	5
4	2122	106.0	100.4	93.8	13.7	5
2	1702	110.5	104.2	97.9	16.3	4

Table 10. Breadth=5, Depth=6.

In table 10 the standard deviation is varied from 2 to 24.

N_{AB}, N, M, N_D, M_L and S are as earlier defined in section 2.4.1.

Graph I.



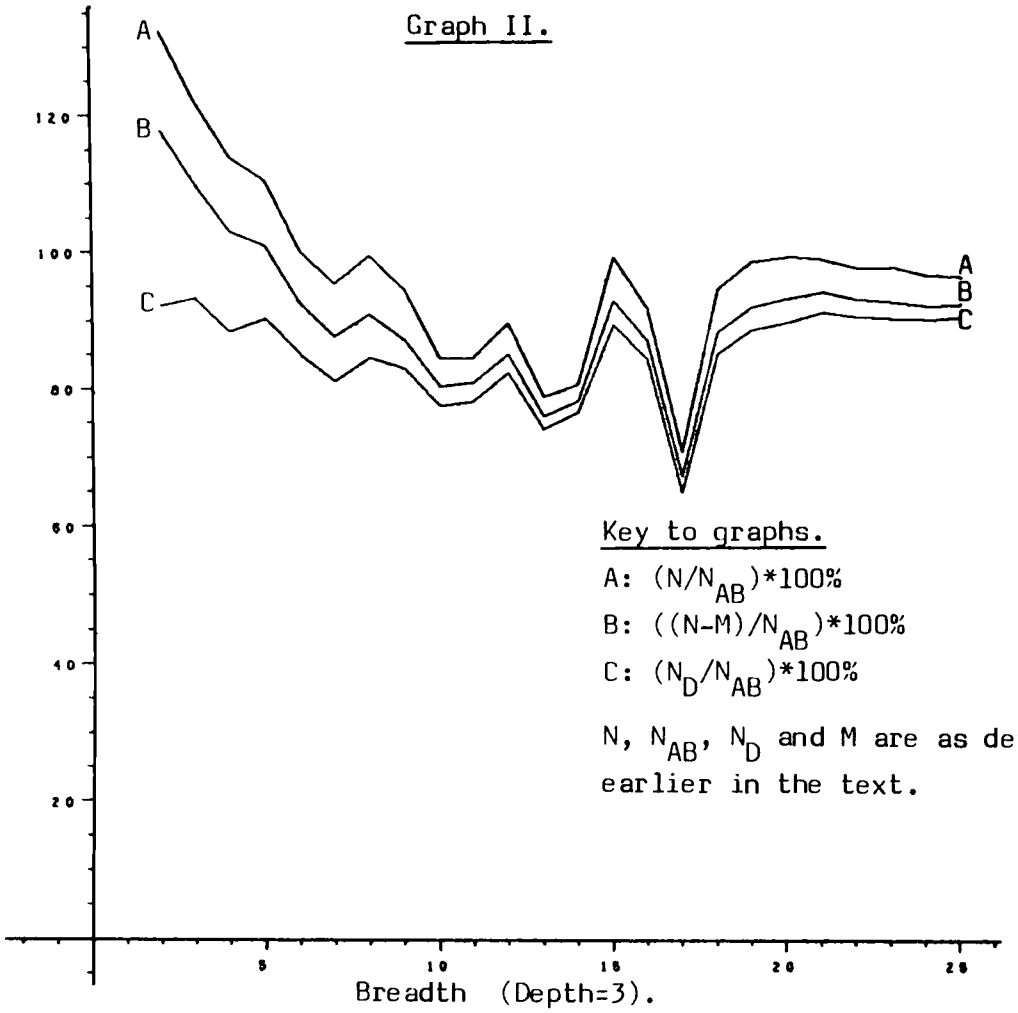
Key to graphs.

A: $(N/N_{AB})*100\%$

B: $((N-M)/N_{AB})*100\%$

C: $(N_D/N_{AB})*100\%$

N, N_{AB}, N_D and M are as defined earlier in the text.



Key to graphs.

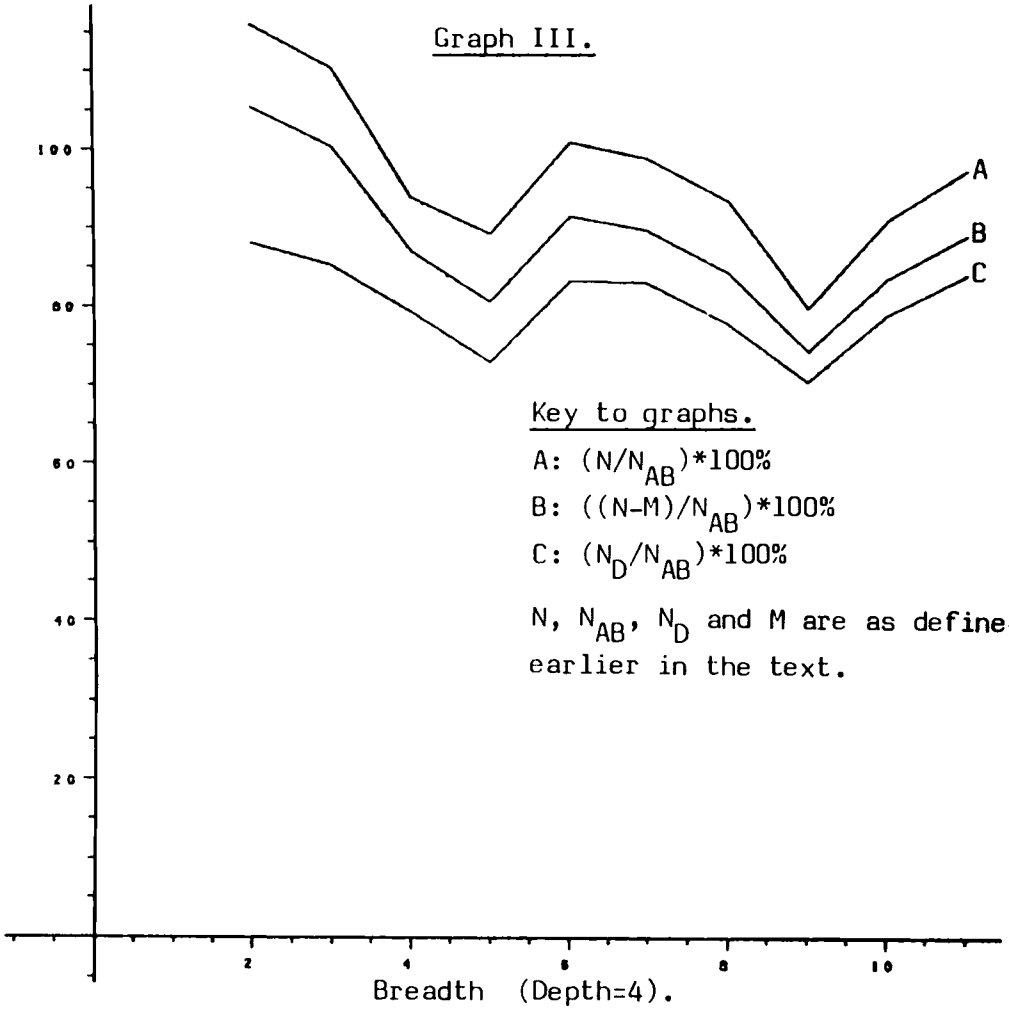
A: $(N/N_{AB})*100\%$

B: $((N-M)/N_{AB})*100\%$

C: $(N_D/N_{AB})*100\%$

N, N_{AB}, N_D and M are as defined earlier in the text.

Graph III.



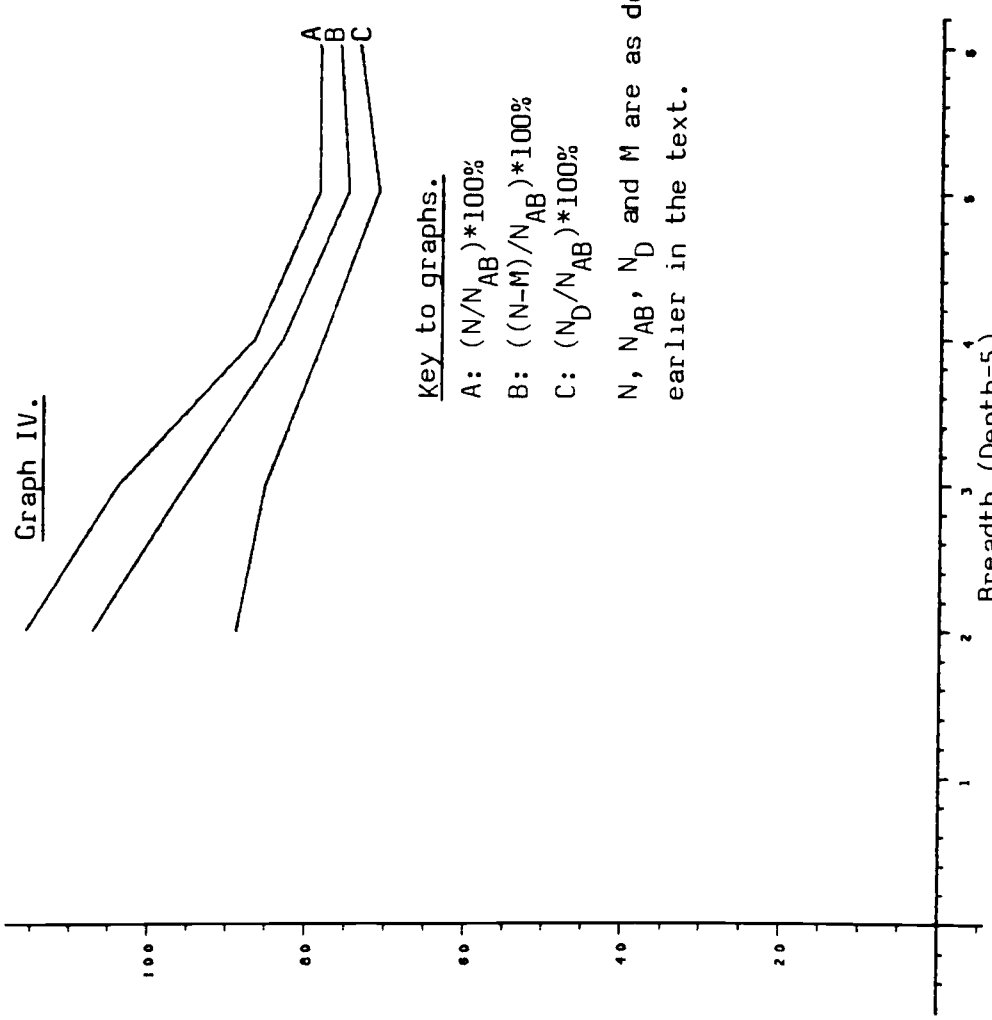
Key to graphs.

A: $(N/N_{AB})*100\%$

B: $((N-M)/N_{AB})*100\%$

C: $(N_D/N_{AB})*100\%$

N, N_{AB}, N_D and M are as defined earlier in the text.


Graph IV.
A
B
C
Key to graphs.
A: $(N/N_{AB})*100\%$
B: $((N-M)/N_{AB})*100\%$
C: $(N_D/N_{AB})*100\%$
N, N_{AB}, N_D and M are as defined earlier in the text.
100
80
60
40
20
1
2
3
4
5
6
Breadth (Depth=5)

2.4.3 Conclusions.

The third column of each of the tables gives the value of $(N/N_{AB})*100$; which is the ratio of the number of nodes looked at by the new-search, including memory reject nodes (i.e those nodes satisfying conditions (i) or (ii) of section 2.3.1) to the number nodes examined by alpha-beta, expressed as a percentage.

The new-search method involves slightly more computation at each node than alpha-beta due to the relative time spent visiting memory rejection nodes and updating memory. However there is an evaluation function to compute at terminal nodes except those which are memory rejection nodes, which is used to derive the numeric backed up values for alpha-beta and newsearch. If this takes a relatively large time to compute, then the time spent on memory rejection nodes, and on updating memory is negligible.

Let T be a game-tree where the set of nodes examined by the newsearch algorithm is a proper subset of the set of nodes examined by alpha-beta.

Let $t1_{new}$ = Total time spent running newsearch on T excluding time spent computing the evaluation function (E.F.).

Let $t2_{new}$ = Time spent computing E.F. during a newsearch run on T.

Let $t1_{ab}$ = Total time spent running alpha-beta on T excluding time spent computing E.F.

Let $t2_{ab}$ = Time spent computing E.F. during an alpha-beta run on T.

Then if t_{new} and t_{ab} are the total times spent on the newsearch and alpha-beta search respectively then:-

$$t_{new} = t1_{new} + t2_{new}$$

$$t_{ab} = t1_{ab} + t2_{ab}$$

If the time taken to compute the E.F. at each node in the newsearch tree is the same as the time taken at the corresponding node in the alpha-beta tree,

then $t2_{new} < t2_{ab}$,

so $t2_{new} = k * t2_{ab}$ for some k: $0 < k < 1$.

$\Rightarrow t_{new}/t_{ab} = (t1_{new} + k * t2_{ab})/(t1_{ab} + t2_{ab})$

So that

$$t_{new}/t_{ab} \rightarrow k.$$

as the time taken to compute the

evaluation function tends to infinity.

Because of this, $(N_D/N_{ab})*100$ is the most important result for comparison. $(N_D/N_{ab})*100$ is displayed in column 5 of each table, and on each graph.

If M_{new} and M_{ab} represent the amount of data storage required for a newsearch and alpha-beta search respectively, then it is not necessarily the case that

$$t_{new} * M_{new} < t_{ab} * M_{ab}.$$

So if

(time spent on search) * (memory requirement)

is taken to be the figure of merit, then the newsearch algorithm does not necessarily perform the most sparing search.

Memory size.

The ratio of the maximum number of nodes stored after any single tree search, to the number of nodes in the alpha-beta tree is shown as a percentage in column 6 of the tables. The results show that the memory-pruning techniques used have been effective in reducing the amount of storage required.

In general, the efficiency of the memory pruning increases with tree size. The greatest efficiency recorded is for trees of depth 3 and breadth 25, where the memory contained at most 1.7% of the nodes searched in the alpha-beta trees.

Pearl [25] has analysed the asymptotic behaviour of random uniform game-trees of depth d and breadth b. In the case that the values at the terminal nodes are just WIN or LOSS such that the probability of WIN at a given terminal node is P0, Pearl shows that using a procedure SOLVE (identical to that used for each single search in our new-search algorithm) as d tends to infinity, the branching factor of SOLVE tends to

$b^{1/2}$ for $P0 = P^*$

$P^*/(1 - P^*)$ for $P0 = P^*$

where P^* is the positive solution of $x^b + x - 1 = 0$. He also shows that for large b, the branching factor is

$$\log(b)/b + O(\log(\log(b))/b)$$

A result that is also shown by Baudet [23]. From this he deduces that as b tends to infinity, the value $P^*/(1 - P^*)$ tends to $b/\log(b)$ and that for $b < 5000$ the formula $(0.925)d^{0.74741}$ is a closer approximation to $P^*/(1 - P^*)$ than $b/\log(b)$.

The values N/N_{AB}, $(N-M) * N_{AB}$ were each found experimentally to have local minima for

depth = 2 breadth = 32

depth = 3 breadth = 10

depth = 3 breadth = 17

depth = 4 breadth = 5

depth = 4 breadth = 9

depth = 5 breadth = 5

No satisfactory reason could be found to explain these minima.

The SCOUT algorithm presented by Pearl [25] also uses the idea of making terminal nodes of trees bi-valued by using an inequality test similar to that which we have described.

The difference between SCOUT and the algorithm presented in this thesis is that SCOUT combines binary-type tree searching with a general evaluation procedure.

Given a MAX node P, SCOUT first calls a procedure EVAL to find the minimax value v(P) of P's first successor P1. Subsequent successor nodes P2,..,Pi,... are then searched with the binary-search procedure to test whether

v(Pi) > v(Pl). If the inequality holds then v(Pi) is evaluated using EVAL and v(Pi) is used for subsequent tests. If the inequality fails then Pi is exempted from evaluation and Pi+1 is tested. A similar test is used when P is MIN node.

CHAPTER 3

The VIRGO chess machine.

3.1. Description of the chess search algorithm.

The Virgo machine uses a version of the search algorithm of chapter 2 to search for wins of material and checkmate in chess positions. The memory described in section 2.3.1 is not incorporated; instead a different memory, described in this section is used to reduce the number of revisited nodes. The values of 1 point for a Pawn, 3 for a knight or bishop, 5 for a rook and 9 for a Queen, are given to the pieces and maximum points are awarded for win of the enemy king.

Modified chess rules

The Virgo system assumes for simplicity of implementation the following modifications to the usual rules of chess.

1) That it is legal for either side to move into check.

2) Accordingly capture of the opposing King constitutes win instead of check-mate.

3) That Pawns may only promote to Queens.

4) Castling is illegal.

5) Capturing Pawns en passant is illegal.

Notation

A move is denoted by an ordered pair [A,B] where A,B are nodes, and B is a successor node of A.

Definition

A node is successful for a given goal iff the backed up truth-value is T for a UTM node or F for a TTM node. A move [P,P1] is successful iff P1 is an unsuccessful node.

In a tactical analysis tree, successful captures are more frequent than successful 1-move threats, because every 1-move threat contains at least one capture. Similarly 1-move threats are more frequent than 2-move threats because every 2-move threat contains at least one 1-move threat. It also takes longer to determine that a move contains a 1-move threat, than that it is a capture, since the former requires a 3-ply search. Similarly moves containing 2-move threats take longer to discover than moves containing 1-move threats.

For this reason each node visited during the search is assigned a cycle number. The cycle number serves to restrict the moves that may be expanded from the node, so that, for example, captures only may be tried on cycle 1, 1-move threats only on cycle 2 and so on. The cycle number is stored for as long as the search is in the scope of the given node and is released when a backup from the node takes place. Each time a node P is visited by descent from its

parent node, the cycle number of P is initialised to 1. If after all successor moves of P have been tried subject to the restrictions of the current cycle number, and no successful move has been found, the cycle number is incremented by one and the successor moves of P are examined again, under the new cycle number. This is repeated until either a successful move from P is found, or a maximum cycle number is reached. In the latter case node P fails. The cycle number imposes the following restrictions on successor moves from a node P:-

Cycle(P) = 1.

In this case only moves which appear to achieve the current material goal in 1-ply are expanded. i.e. moves for which

tally(P+) $>$ tally(P) P a UTM node

tally(P+) $\leq$ tally(P) P a TTM node.

Where tally is defined by;

tally(P0) = 0

tally(P+) = tally(P) + (increase in material due to captures and promotions) * i

i = 1 for P a UTM node.

= -1 for P a TTM node.

And where G is the current goal.

Cycle(P) = 2.

In this case the moves expanded are

a) Those moves expanded under cycle(P) = 1.

b) Moves containing a 1-move threat.
The definition of a 1-move threat is given below.

Cycle(P) = 3.

In this case the moves expanded are

a) Those moves expanded under cycle(P)=2.

b) Moves containing a 2-move threat. The definition of a 2-move threat is given below.

Let P be a node with first predecessor P-1 and second predecessor P-2. Suppose cycle(P-2)=cycle(P)=n say, and that all successor moves of P have been tried and have failed. It would be illogical to immediately set cycle(P)=n+1, because not all the relatively simple cycle-n moves from P-2 need yet have been examined. For this reason cycle(P) is never allowed to exceed cycle(P-2) so that if cycle(P)=cycle(P-2) and no successor move of P is successful, then P fails.

Within each cycle, the candidate moves are divided into six types. These are:-

1. threat
2. dummy
3. memory
4. recapture
5. consequent
6. general

Some of these types may be illegal but those types that are legal are always tried in the order above. e.g. threat moves before memory moves, and recapture moves before general moves.

One-move threats.

Let P be a node for which cycle(P)=2 and let [P,P1] be a move that did not qualify as a cycle 1 move. Then [P,P1] must be tested, to determine whether it contains a 1-move threat. The search first moves from P to P1, whereupon a dummy move is tried at P1. A dummy move is a move that takes the search one ply deeper, but does not alter the current board position. Let the dummy move be [P1,P2]. Cycle(P2) is set to 1, and a search commences for a successful move from P2. If none is found, cycle(P2) is not incremented, and the search backs up to P; the move [P,P1] having failed. If a successful move is found however, the move is stored as a threat move; and the search backs up to P1, whereupon a search starts from P1 for a defence to the threat. As long as the search remains in the scope of P1, the threat move is always tried first, from all successor

nodes of P1 for which it is legal (see figure 1). A threat move is stored as a given piece moving in a given direction, a given number of squares.

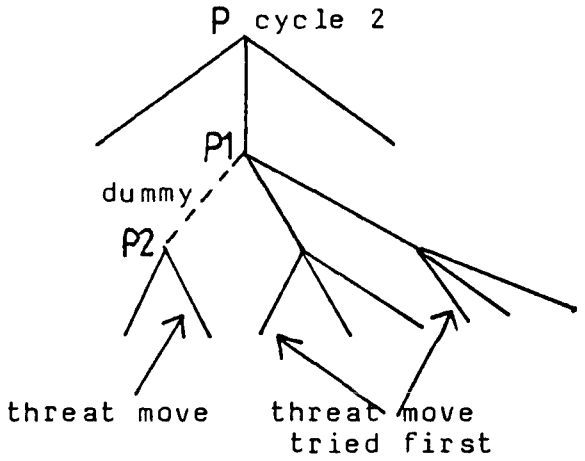


Fig 1.

Two-move threats.

Now suppose P is a node for which cycle(P)=3 and let [P,P1] be a move that did not qualify as a cycle 2 move. Then [P,P1] must be tested to determine whether it contains a two-move threat. A dummy move [P1,P2] say, is tried at P1. Cycle(P2) is set to 1, and a search commences for a successful move from P2. If none is found, cycle(P2) is incremented to 2, and the search continues from P2. If no successful move is found, cycle(P2) is not incremented to 3, but the search backs up to P. If a successful move is found, then a search starts from P1 for a defence to the threat.

Defences.

Let P be a node for which cycle(P)=2 and let [P,P1] be a

move that did not qualify as a cycle 1 move. Now suppose that after trying the dummy move [P1,P2] the threat move [P2,P3] is found. Then a search must start from P1 for a defence to the threat. Since [P,P1] did not qualify as a cycle-1 move, then if

a) P is a UTM node, then

   tally(P1) $\leq$ goal and tally(P1+) $\leq$ tally(P1)

   for all P1+ successor nodes of P1.

   => tally(P1+) $\leq$ goal.

b) P is a TTM node, then

   tally(P1) $>$ goal and tally(P1+) $\geq$ tally(P1)

   for all P1+ successor nodes of P1.

   => tally(P1+) $>$ goal.

In both cases all successor moves of P1 qualify as cycle 1 moves.

In order to try first the defence moves from P1 which are most likely to succeed and require the least computation, the following restriction is imposed on cycle-1 moves from P1.

The move [P1,P1+] is tried as a cycle-1 move iff, either the threat move is not a legal successor of P1+, or the threat move does not qualify as a cycle-1 move as defined earlier

in this section.

In this case [P1,P1+] is called a cycle-1 defence.

Cycle-2 and cycle-3 defence moves.

Suppose cycle(P1)=2 or 3, and [P1,P2] is a move that did not qualify as a cycle-1 defence. Let [P2,P3] be the threat move. To determine whether [P1,P2] is a cycle-2 or 3 defence, a search is made from P3 for a successful move. During this search, cycle(P3) may never exceed cycle(P1)-1. [P1,P2] then qualifies as a defence iff a successful move from P3 is found.

Memory moves.

Let [P1,P2] be represented by the ordered pair (pc1,sq1), where pc1 is the piece moved and sq1 the square it was moved to. Let [P2,P3] be represented by the triple (pc2,dir2,n2), where pc2 is the piece moved, dir2 the direction it was moved in, and n2 the number of squares it was moved. If [P2,P3] was successful then the move (pc2, dir2, n2) is stored in an array "mem" so that mem(pc1,sq1)=(pc2,dir2,n2). Now suppose that at some later point in the search, the move [P1',P2'] is tried. If this move is represented by (pc1',sq1') where pc1'=p1 and sq1'=sq1', then the MEMORY MOVE associated with [P1',P2'] is mem(pc1',sq1') mem(pc1,sq1) = (pc2,dir2,n2), so that at the node P2' the move (pc2,dir2,n2) will be tried as a memory move. Frequently, a successful reply to a move made at one point

in the search tree will have the same reply at a different point. Now only a threat or dummy move can be tried before a memory move, so that the memory move, if there is one, is tried early in each cycle. This has the effect of reducing the amount of repeated analysis made during the search. Incorporating memory moves into the search algorithm is particularly advantageous in view of the fact that many nodes are revisited during the search.

Recapture moves.

One of the most frequent replies to a capture or threat is to capture the piece that made the capture or created the threat.

If [P1,P2] is a move of piece pc1 say, then a move [P2,P3] that captures pc1 is called a RECAPTURE MOVE. Recapture moves are tried after the memory move in each cycle. It takes less time to compute the recapture moves than the remaining moves for the following reasons:-

a) If [P2, P3] involves moving piece pc2 say, then it may be possible to decide at once that pc2 cannot capture pc1 given the coordinates of pc1. e.g. If pc2 is a rook, then pc1 must share pc2's x or y coordinate.

b) Given that pc2 is not rejected by a) above, then it is possible to determine the direction pc2 must move in, to capture pc1, from the coordinates of pc1 and pc2, without

running through every legal direction for that piece.

Consequent moves.

Let [P1,P2] be a cycle-2 move of piece pc1 in direction dir1. Let [P2,P3] be a dummy move. Then a frequent one move threat [P3,P4] involves a capture by piece pc1, where the capture was not possible before the move [P1,P2]. Such a move, [P3,P4] is called a CONSEQUENT MOVE. When cycle(P1) = 2 then only consequent moves are legal at node P3.

General moves.

All moves that have not already been tried under the previous types are called GENERAL MOVES.

Limiting the search.

If the cycle number was never allowed to exceed 1 for both sides, then each search would be a capture chain and so would terminate. However threat chains do not necessarily terminate, so a restriction must be made on the total number of moves that are expanded. To do this a count is made of the number of moves likely to be expanded, before entering cycle-2 or cycle-3. The result of this count is called the brake number. The product is now formed of all the brake numbers of ancestor nodes to the current node.

i.e. product(P0) = 1 if brake(P0) = 0

= brake(P0) otherwise.

product(P) = product(P-) if brake(P) = 0.

= brake(P) * product(P-) otherwise

If product(P) exceeds some fixed limit L say, then the search is prevented from going more than a few ply deeper than P, by the following procedure.

For all P' which are UTM nodes in scope of node P, cycle(P') must not exceed 1 and if [P',P'+] is a defence move, it may only be tried if it is a capture.

Now suppose P0 has just one successor P1 that needs expanding so that brake(P0) = 1, and product(P0) = 1. Further suppose P1 has just one successor node P2 that needs expanding so that brake(P2)=product(P2)=1. If this continues for nodes P2, P3, ... without limit, then if $L \geq 1$ the search will not terminate. To prevent this a depth limit D may be set so that if the depth exceeds D then the search terminates within a few ply of depth D, in the manner described above.

Iterative deepening

At the start of each search with a given goal, the brake limit is set to 1. If the search fails, the goal remains the

same but the brake limit is set to 2 and the search is repeated. The brake limit is incremented in powers of two up to a maximum of 256. If no successful move has then been found the search for that goal fails and a new goal is set. Incrementing the brake limit in this way, reduces the search for a successful move when a simple solution exists, but increases the search time when a successful move is not found until the last iteration.

Further search simplification.

If the search is in the scope of two dummy moves played by one side (A say) then moves played by side B are only considered good to win any material if they are King captures or Pawn promotions.

This simplification enabled special circuits to be constructed to detect certain cases when there are no King captures or Pawn promotions, in order to reduce search.

3.2 The hardware.

The algorithm described in the previous section was implemented in hardware consisting of 500 logic integrated circuits mounted on five wire-wrap boards. Plate 1 shows the completed machine mounted on its 3' 7" x 3' 2" x 9" box. Plate 2 shows the wiring side of the logic boards and plate 3 the logic integrated circuits. The machine operates in either stand-alone mode or interfaced to a VAX 750 mainframe via an LSI-11/23 computer.

In stand-alone mode, input is through a pad of twenty push buttons keys for inputting positions, setting parameters and running searches. Output is sent to a thirty-two character display.

The parameters that may be set are:-

1) The maximum cycle number (to 1, 2 or 3).

2) The maximum brake number (from 2 to 256 ascending in integral powers of 2).

3) The depth limit, which forces termination within a few ply (from 1 to 255).

The interface to the mainframe is described in section 3.6.

The machine is driven from a clock that has two cycle types

a) A fast cycle for evaluating terminal nodes.

b) A slow cycle for descending or ascending moves (130 KHz)

The machine typically evaluates 5.5 times as many terminal nodes as non-terminal nodes and generates 1 legal move for every 2.5 candidate legal moves which include some illegal moves such as moving a piece beyond the edge of the board.

The core of the VIRGO hardware is illustrated in the block diagram of figure 2.

The position memory.

The position memory is divided into two units. The first unit outputs a square for a given piece, while the second unit outputs a piece for a given square.

The piece-square memory.

This consists of 32 locations, one for each of the 32 chessmen, each location holding an 8-bit binary word. The first bit is a 1 iff the respective piece is on the board. Three bits give the x-coordinate of the piece, and a further three bits give the y-coordinate. The remaining bit is used for the pawn locations only. This is a 1 iff the pawn has been promoted to a queen.

The square-piece memory.

This consists of 64 locations, one for each square of the chessboard. The memory is addressed with the X-Y coordinates

of a square, and outputs the contents of that square in a six-bit binary word. One bit specifies whether a piece occupies the square; the other five bits give the code for the piece on the addressed square if there is one.

The move generator.

The move generator consists of five counters:

1) Number of squares counter: This is a three-bit counter giving the number of squares a piece is to move.

2) Direction counter: a 3-bit counter giving the direction in which a piece is to move.

3) Piece counter: a 4-bit counter giving the piece being moved.

4) Type counter: a 3-bit counter giving the type of move being made (threat, dummy, memory, etc.).

5) Cycle counter: a 2-bit counter giving the current cycle number.

The piece counter addresses the piece-square memory which supplies the X and Y coordinates of the piece. These are called the "oldsquare coordinates". The 3-bit direction counter and 4-bit piece counter are fed into an encoder that produces a 4-bit word representing the direction the piece moves in. The oldsquare coordinates, the four-bit direction code, and the number of squares counter are all fed into the

newsquare unit. This supplies the coordinates of the square the piece is to move to.

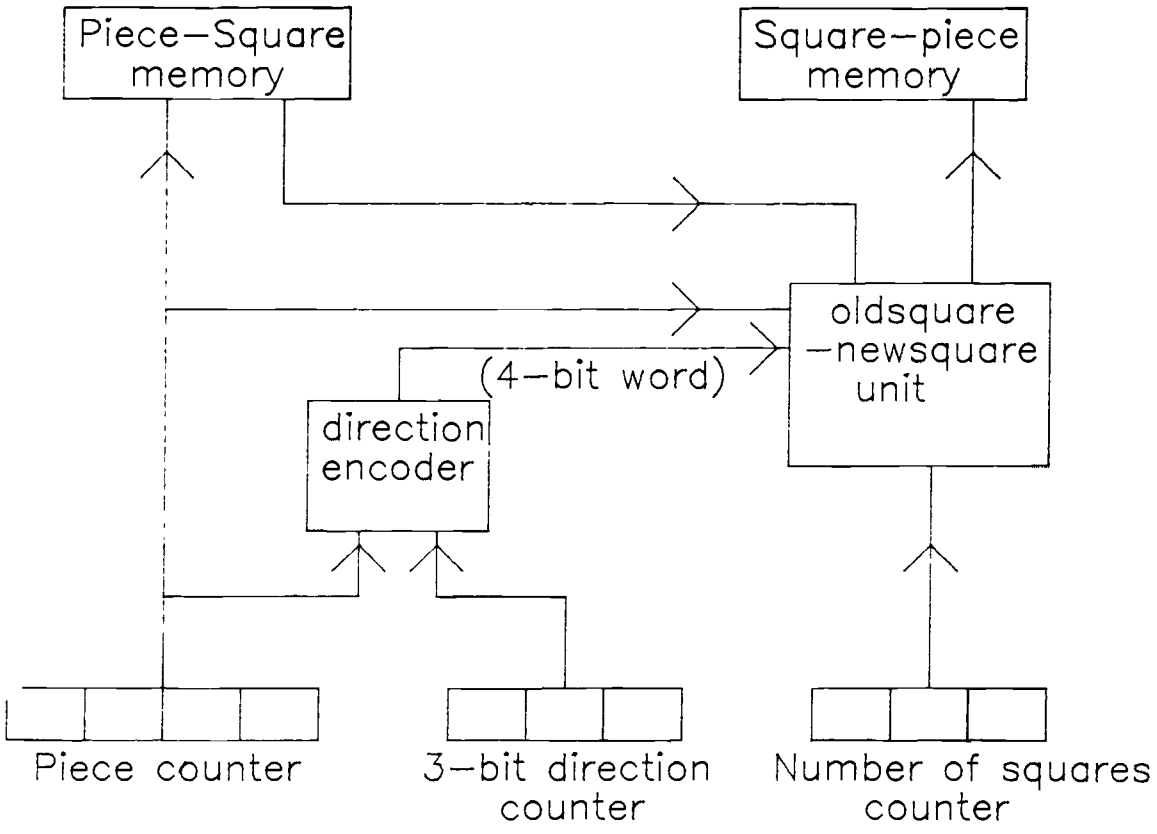
Finally the newsquare coordinates address the square-piece memory, which outputs the piece on the newsquare. If this is of the same colour as the piece being moved, then the move is illegal. If this is of opposite colour to the piece being moved, then the output piece is captured.

The stack.

Each time a descent is made in the analysis tree, the piece moved, direction, number of squares, type, cycle number, etc. need to be stored to enable the backing up of moves. All this information is stored in a LIFO stack in the form of an 80 bit word. The stack is made from 256 x 4 bit random access memories, so that there is sufficient storage to permit the analysis to go 256 ply deep. Obviously this does not happen in practice.

FIGURE 2

Figure showing the core of the VIRGO hardware.



Other chess hardware.

Moussouris, Holloway and Greenblatt [14] built a chess machine "CHEOPS" for rapid alpha-beta search, based on a special processor for executing chess algorithms. Condon and Thompson [8] constructed the "Belle" chess machine that performed rapid alpha-beta search and also included an evaluation function, and a transposition table for enabling extra tree pruning.

Schaeffer Powell and Jonkman [18] have more recently implemented a legal move generator in VLSI circuits.

Interface of search machine to VAX/750 computer via an LSI-11/23 computer.

The search machine is interfaced to a Digital LSI-11/23 computer, through a 48 line parallel port.

The machine outputs 256 bits, through 8 bits on the output port multiplexed 32 ways. 20 bits on the port are used for simulating the action of the push buttons, and a further 11 bits are used to enable interactive control over the search. The remaining bits control multiplexing of the machine's output. The LSI running a reduced version 6 Unix, from John Hopkins University, services low-level commands to the search machine. The LSI in turn is interfaced to a VAX 750 main frame, through a 9600 baud serial line. The following are some of the routines that allow programs on the VAX to communicate with the search machine.

to_move(m):

Get to move m, at current node, where m is a move descriptor specifying moved piece, destination square and cycle number. Return 1 if move is found, 0 otherwise.

to_suc_move(m)

Move descriptor m is loaded with the first successful move found by the machine at the current node, if there is one. The routine returns 1 if a successful move is found 0 otherwise.

all_suc(ma,n)

Loads move array ma with up to n successful moves at the current node. The routine returns the number of successful moves found.

up(n)

Force back up through n ply if currently at a depth of at least n.

down() If at a legal move, go down it

find_best_move(iteration,goal,m)

The search machine seeks a solution to a previously loaded position and loads move descriptor m with the "best" first move of the combination. The integer variables iteration and goal are loaded by the routine with the iteration and goal values at which the best move was found.

get_machine_state(s)

The state descriptor s is loaded with information about the current state of the machine.

load(P)

The search machine is loaded with position P.

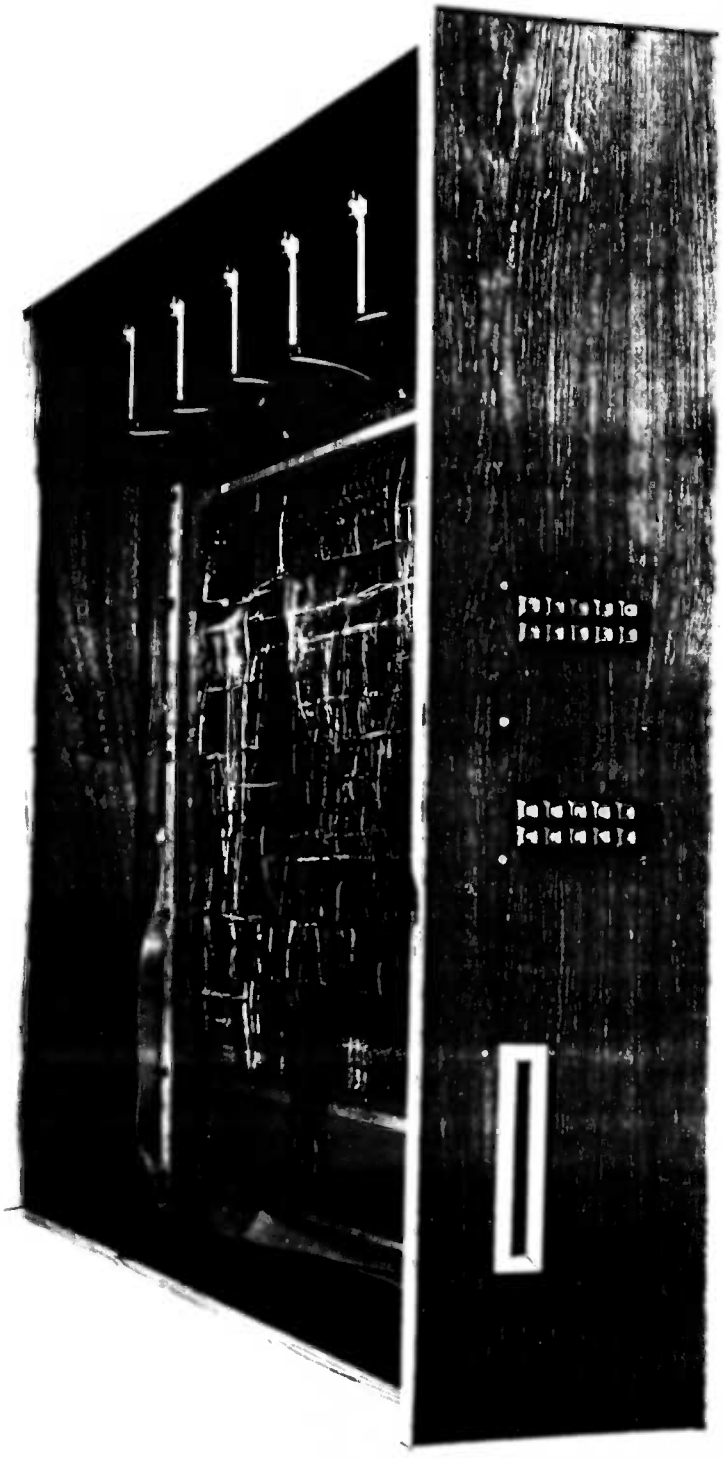


Plate 1 showing the front view of the VIRGO chess machine.

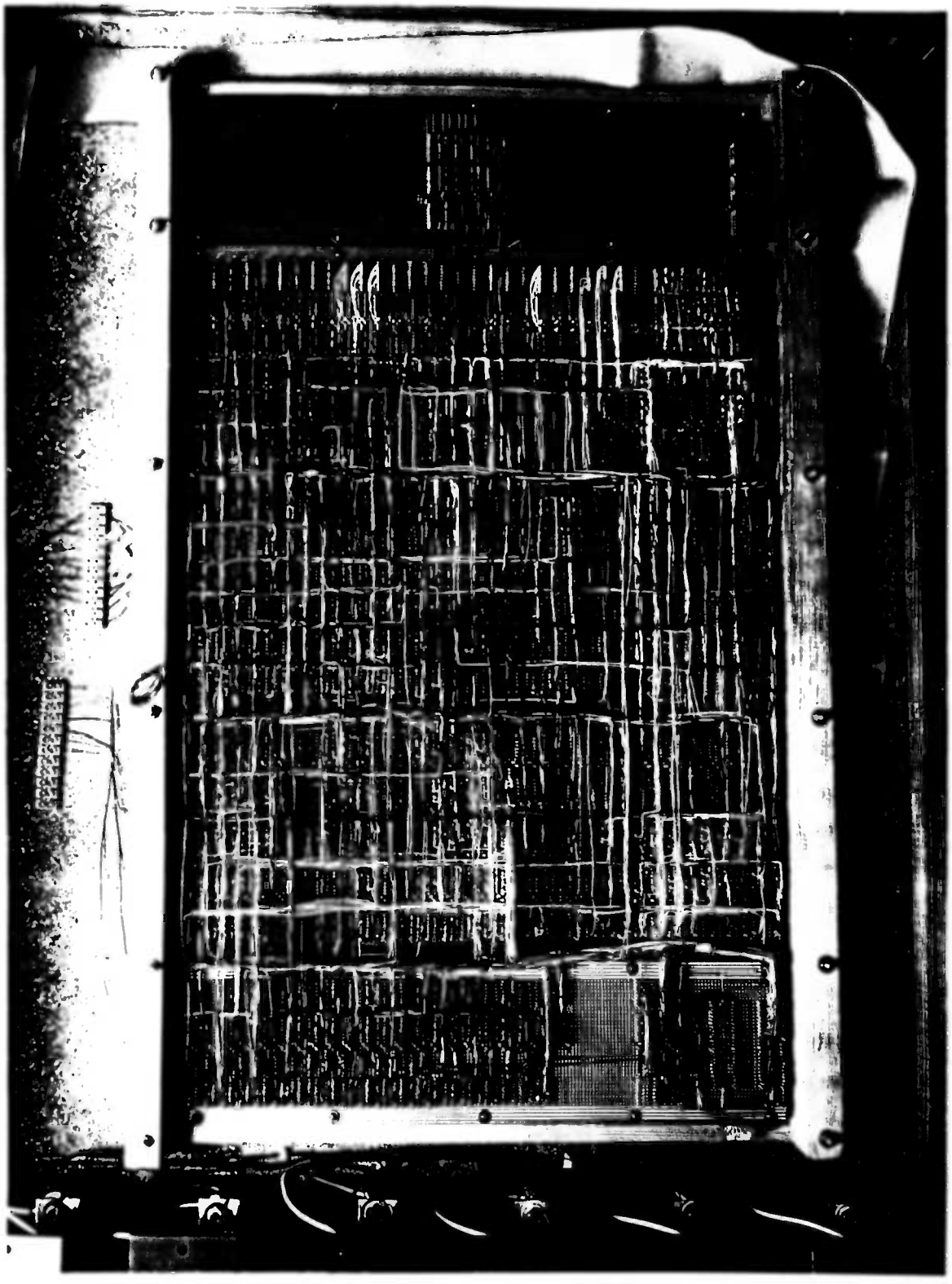


Plate 2 showing the wiring side of the logic circuits in VIRGO.

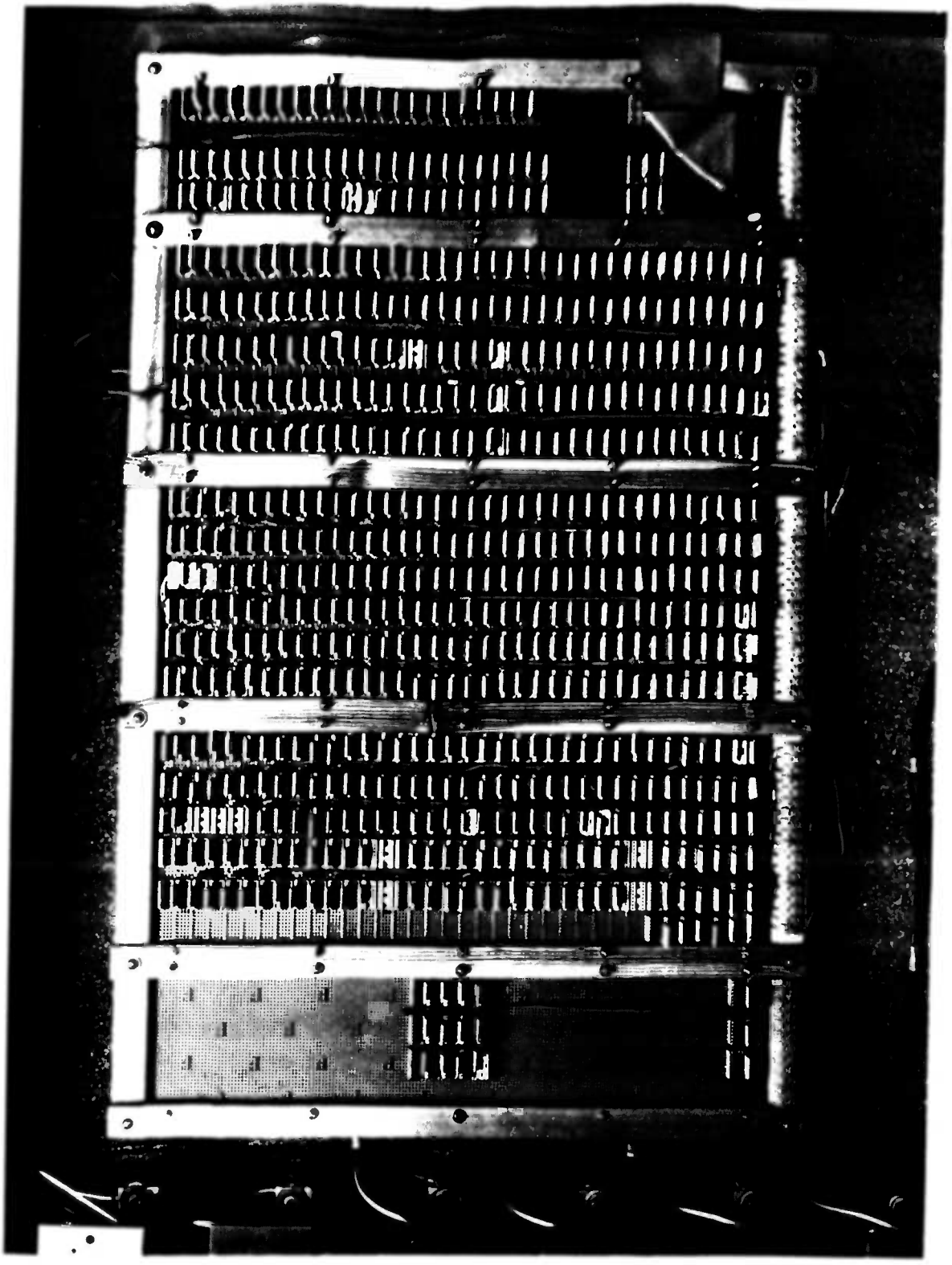


Plate 5 showing the logic circuits in VIRGO.

3.3 Experimental testing of the hardware.

VIRGO was tested on 30 positions taken from all of chapters 3, 8 and 9 of "Chess Positions" by C.H.O.D. Alexander [1]. Throughout these tests the depth was set to 14 ply, and the maximum iteration no. to 8. Each position was tested with

A) The maximum cycle no for UTM nodes set to 3, and the maximum cycle no for TTM nodes set to 2.

B) The maximum cycle no for both UTM and TTM nodes set to 2.

A time limit of 15,000 secs was set for each position. After that time had elapsed the last successful move found, if any, was counted as VIRGO's move for that position.

3.4 Results.

Test A)

For 18 positions solved, VIRGO's first move agreed with the move given by Alexander and the positions completely solved within the time limit. For 2 positions the time limit elapsed before VIRGO completed its search, but the move returned agreed with Alexander's move. This gives 20 positions out of the 30 tested correctly solved.

For 3 positions, VIRGO found an alternative first move to the one suggested by Alexander, for which it is unclear whether or not VIRGO's move is as good as Alexander's.

VIRGO failed to correctly solve the remaining 7 positions. 3 of these were time failures, 3 others were of an endgame type, beyond VIRGO's tactical search. The last remaining position was unsolved because it contained a two-move threat (cycle-3 move) that did not involve capture of the opponent's King.

The mean time taken to solve these positions was 6446 secs (=1hr 48mins).

If the first win of material by VIRGO, instead of the highest valued win is counted as being VIRGO's move, then on 19 of the 30 positions, VIRGO's move would have been the same as as Alexanders. The mean time for solving each position would then be 2436 secs/position (=40mins 36secs/position).

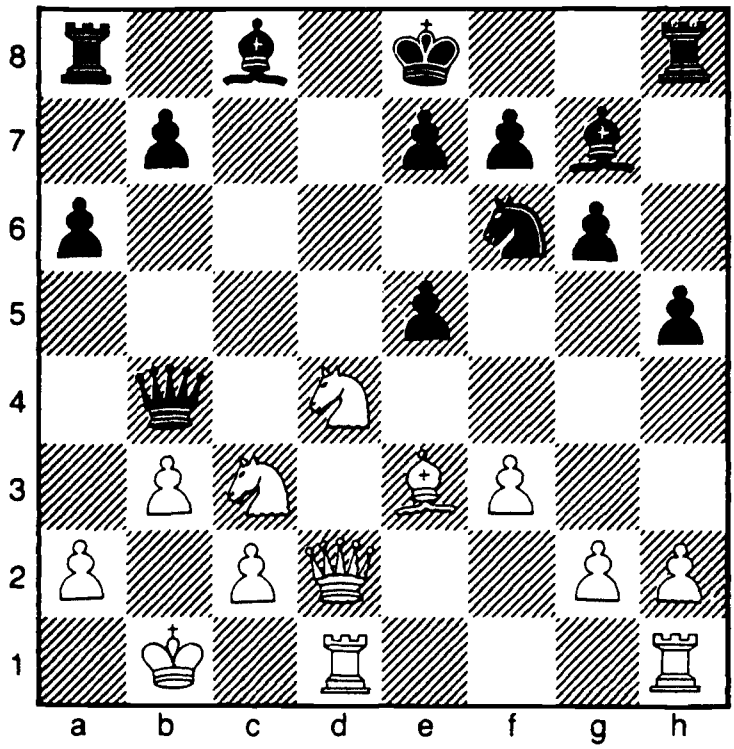
VIRGO searches a mean number of 90,000 nodes/second.

Test B)

When VIRGO was set to only consider cycle-2 moves, then VIRGO correctly solved 12 positions and failed to solve 18 positions. The mean time to solve each position was 774 secs/position (=12mins 54secs/position).

Detailed timings, for test A, for runs on six of these positions are tabled beneath each diagram below. Each line of the tables shows the time in seconds for VIRGO to run a search on one goal setting, the goal settings being listed

sequentially (top down) for each position. The iteration entry is in each case the maximum iteration number attained for each goal. Whenever a search on a given goal setting yielded a successful move, then the move and cycle no are entered. Otherwise the appropiate table entry is a blank. After each table the value VIRGO returned for material gain, together with the last successful move found are entered. For the first five of these positions the move given by VIRGO agreed with the move given by Alexander. The last position (no 5 of chapter 8 in [1]) is from an endgame and beyond VIRGO's capacity. Alexander gives the winning line 1. Kc4 then if 1...Ke4 White can Queen his a-Pawn, and if 1...Pf3 then 2.Pxf3, Pxf3 3.Kd3!



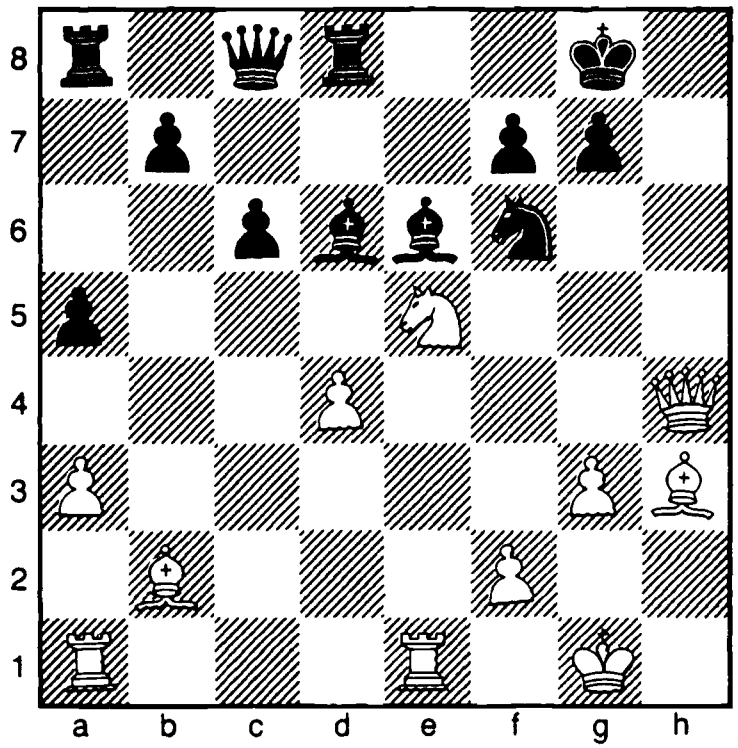
goal	iteration	move	cycle no	time(secs)
1	3	N(d4)b5	3	4585
33	8			126
7	8			2672
4	7	N(d4)b5	3	3876
5	8			3876

material gain = 4

best move = N(d4)b5 on cycle 3

total time = 14513 seconds

Table showing the result of test A on position 3, chapter 10 of "Chess Positions" (shown above).



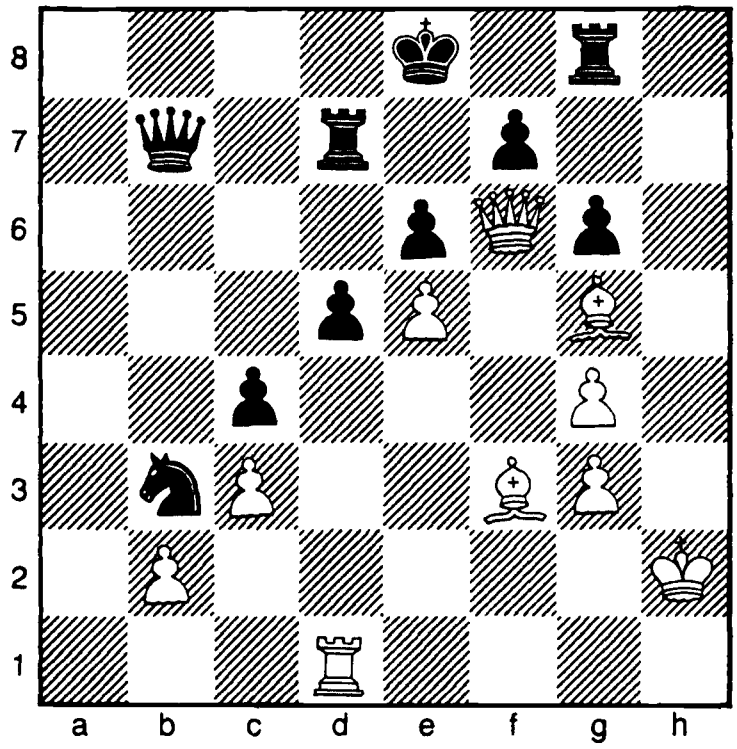
goal	iteration	move	cycle no	time(secs)
1	1	Nxf7	1	5
33	8			78
13	8			102
7	8			729
4	8			5772
2	3	Nxf7	2	1151
3	4	Ng6	3	5719

material gain = 3

best move = Ng6 on cycle 3

total time = 13512 seconds

Table showing the result of test A on position 5, chapter 10 of "Chess Positions" (shown above).



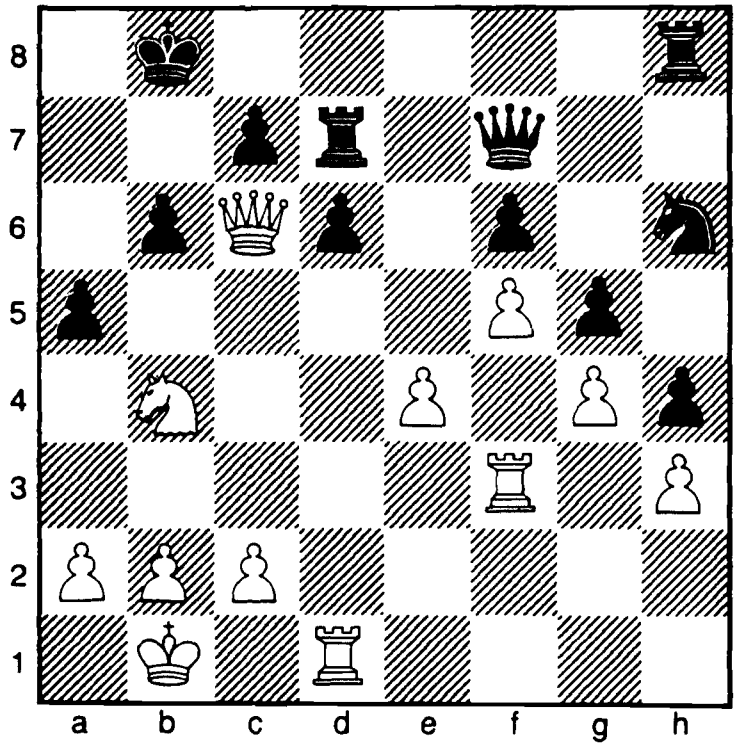
goal	iteration	move	cycle no	time(secs)
1	4	Bxd5	3	567
33	8			56
9	8			81
5	8			136
3	8			134
2	8			680

material gain = 1

best move = Bxd5 on cycle 3

total time = 1654 seconds

Table showing the result of test A on position 6, chapter 10 of "Chess Positions" (shown above).



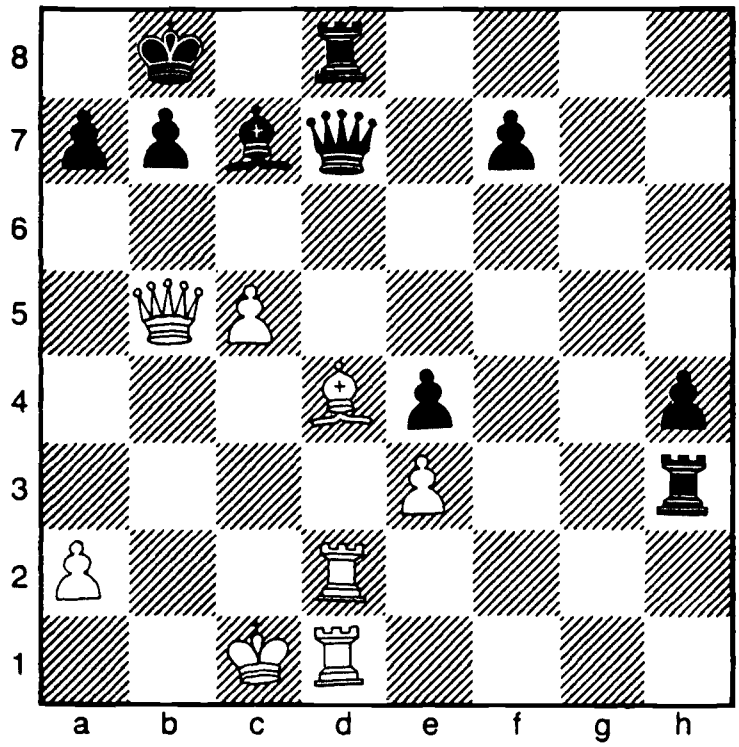
goal	iteration	move	cycle no	time(secs)
1	5	Na6ch	3	3128
33	8			140
8	8			203
4	6	Na6ch	3	2064
6	8			1562
5	8	Na6ch	3	2975

material gain = 3

best move = Na6ch on cycle 3

total time = 7072 seconds

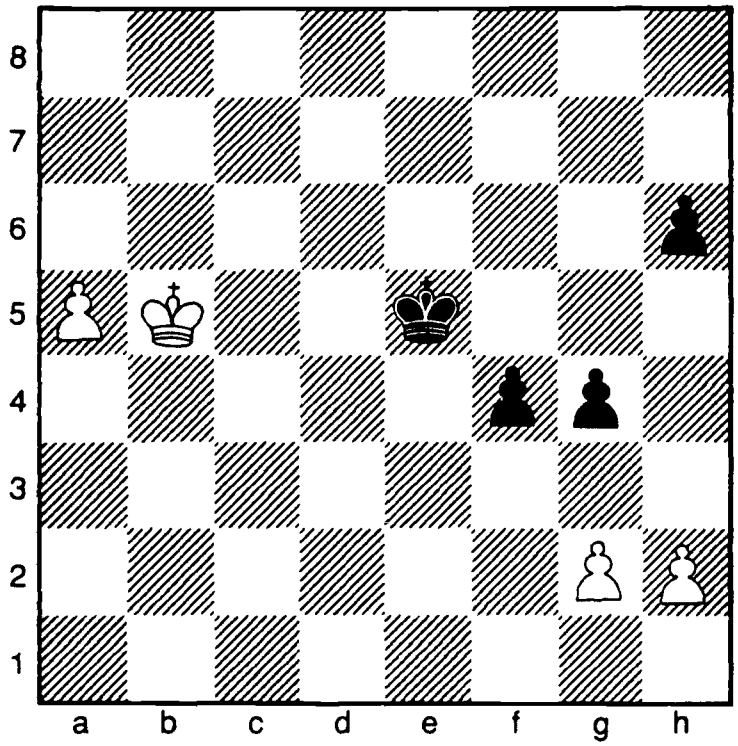
Table showing the result of test A on position 7, chapter 8 of "Chess Positions" (shown above).



goal	iteration	move	cycle no	time(secs)
1	1	Be5	2	4
33	8			69
10	8			701
5	8	Be5	3	5996
7	8			4850
15,000 second time limit expired.				

best move = Be5 on cycle 3

Table showing the result of test A on position 3, chapter 8 of "Chess Positions" (shown above).



goal	iteration	move	cycle no	time(secs)
1	8			3
-31	1	Kc6	1	0
0	1	Kc6	1	0

material gain = 0

best move = Kc6 on cycle 1

total time = 3 seconds.

Table showing the result of test A on position 5, chapter 8 of "Chess Positions" (shown above).

3.5 Deficiencies and solutions.

A number of modifications could be made to the hardware that would result in shorter search times.

1) VIRGO does not store pairs of backed up values as described for the search algorithm in Chapter 2. If memory was incorporated then extra cutoffs could be made at revisited nodes and some large subtrees would not searched. The overheads involved are

   1) The cost of hardware for storing trees.

   2) The total time spent accessing memory.

   Both these overheads could be reduced by limiting the depth to which nodes are stored.

   The Belle chess hardware [8] uses an associative memory to recognize positions that have already been analysed at different nodes in the search tree but are the same by transposition. The associative memory is then used to make extra alpha-beta cutoffs. The inclusion of an associative memory in VIRGO would reduce the number of nodes visited during searches.

2) Further move orderings would be desirable in addition to the division of moves into cycles and move types. It is possible for example for the machine to be searching a node where the opponent's King, or a valuable piece such as the Queen is en prise, but waste time examining

moves other than the obvious capture.

A static ordering that, for example, placed moves in order of greatest immediate capture value would prevent wasted search such as described above. The overhead involved in doing this is the cost of memory for storing moves at the current node, and then stacking them while searching subtrees of descendant moves. The overhead of extra time, needed to make this ordering, could be avoided by making the ordering during cycle 1, and using it during cycles 2 and 3.

3) Finally it may be possible to further reduce search times by sacrificing a knowledge of the upper bound of the material gain. Once a successful move has been found at the root node, while it remains the current best move, there is no need to re-examine it on subsequent searches. If all the other moves fail on a subsequent search then the search may halt with the current best move as the final best move. This can save re-examining the same move at the root node, where the only gain in information is the maximum material that can be won by that move. If however another move is successful on a subsequent search, then the new move supersedes the old one as the current best.

CHAPTER 4.

The explanation facility.

4.1 Overview of the explanation facility.

The VIRGO machine given a position simply returns a single move. This does not necessarily give the user any idea of why the move was chosen. Outputting the forcing tree searched by the machine would not help because it may have thousands of branches and be unintelligble to the user. However the solution to chess positions can be explained to humans in terms of certain elements called "pins", "decoys", "overloads" etc. These elements or "motifs" can be extracted from chess positions by using the search machine in a process of making causal tests on moves in its search tree.

Note

Throughout this chapter the VIRGO search machine is referred to as SM.

We will call the algorithm that makes these causal tests the alternating search routine (ASR). The ASR is presented with two UTM positions P1, P1' say, for which a successful move can be found at P1. The SM is set to find the move [P1,P2] that was successful at P1. [P1,P2] can be represented by the ordered pair (pc,sq) where pc is the piece moved and sq the destination square. Assume that (pc,sq) is a legal unsuccessful move [P1',P2'] say, at position P1'. The ASR

then asks the SM why the move [P1',P2'] was unsuccessful. The SM may answer this with a move [P2',P3'] = (pc2,sq2) say. The ASR then tests whether (pc2,sq2) is legal in position P2 and the process continues until either

a) A move is illegal in one of the lines and so cannot be tried

or b) A move is found that wins enough material to immediately achieve the goal in one line, but not the other.

If a record is made of these two lines of play together with the reason for the last move tried working in one line but not the other, the record constitutes a description of the "cause" of move m1's success. This method has the merit of reducing large and awkward trees to pairs of move chains.

4.2 Two examples.

Example 1.

Consider the position shown in figure 1 below.

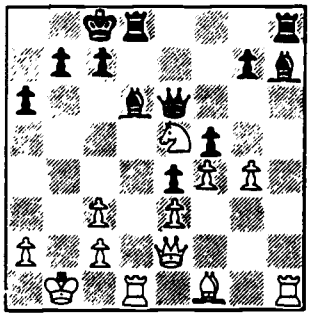


FIGURE 1.

White can win material by

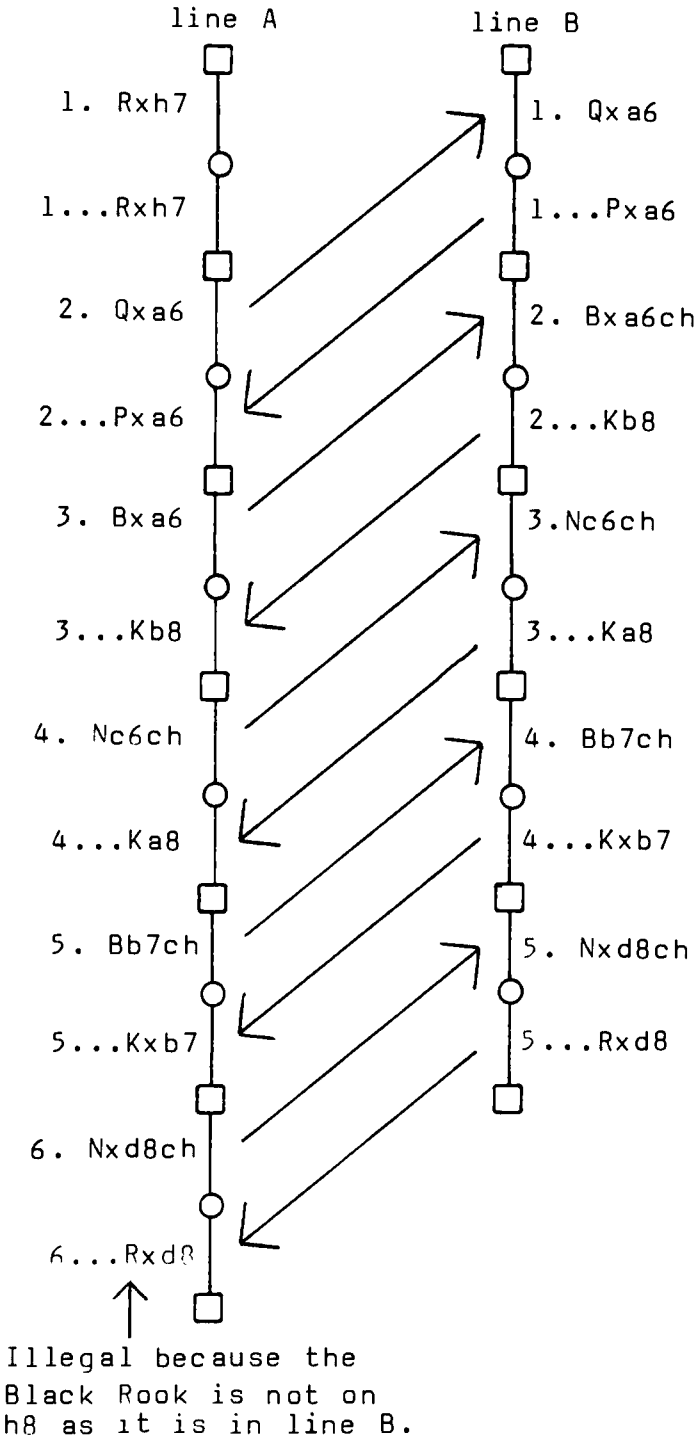
1. Rxh7 Rxh7
2. Qxa6 Pxa6
3. Bxa6ch Kb8
4. Nc6ch Ka8
5. Bb7ch Kxb7
6. Nxd8ch Kc8
7. Nxe6

The combination is based on an overload. The Black Rook on h8 defends both the Bishop on h7 and the Rook on d8. After 1. Rxh7, Rxh7 the Black Rook (h7) no longer defends the Black Rook (d8). However it is not apparent that the Black Rook (d8) needs defending until White plays 6. Nxd8ch. Discovering that the combination contains this overload

involves asking the question "What is the reason for white's move 1. RxB ?" The answer can be found by seeing what happens if White omits this move and plays his second move first instead; i.e. White plays 1. Qxa6. We now have two lines of play. The first beginning 1. Rxh7, Rxh7 2. Qxa6 and the second omitting 1. Rxh7 but commencing 1. Qxa6. We now ask the question "What would Black's best response be to 1. Qxa6 in line B. The answer is 1...Pxa6. We now return to line A and ask the question, "What would White's response be to 1...Pxa6 ?" The answer is 3. Bxa6ch. We now apply this move back to line B and continue to alternate between the two lines until the move 5. Rxd8, when applied back to line A, is found to be illegal (see figure 2). It is now apparent that the move is illegal in line A because the Black Rook is on h7 and cannot move to d8, but legal in line B because the Rook is on h8 where it can move to d8 and capture the White Knight.

FIGURE 2

Figure showing the two lines of play that indicate the decoy of the B. Rook from h8 to h7 in the position shown in figure 1.



Example 2.

Consider the 4-move problem in figure 3 below.

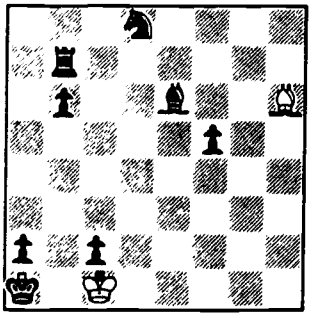


FIGURE 3.

The solution is:-

1. Bg5 (threatening Be6mate) Rf7
2. Bf4 (threatening Be5mate) Nc6
3. Bd2 (threatening Bc3mate) any
4. Bc3 mate.

We wish to discover the reasons for 1. Bg5 and 2. Bf4 being necessary. To do this we use the same technique as before; i.e. we see what happens when each of these moves is omitted.

We first examine the two lines of play resulting from

a) Playing 1. Bg5

b) Omitting 1. Bg5 and playing the move specified as White's second move, i.e. Bf4, instead. (see figure 4.)

In line B, 1...Nf7 is Black's response to 1.Bf4. So we try

this move in line A after 2. Bf4. However it is illegal in line A because Black's Rook occupies f7. It is now apparent that 1. Bg5 is necessary to draw the Black Rook to f7 where it blocks Black's Knight.

To discover the reason for 2. Bf4 we consider the two lines of play resulting from

a) Playing

1. Bg5 Rf7

2. Bf4 Nc6

b) Omitting 2. Bf4 but playing the move specified as Whites third move; i.e. 2. Bd2 instead. (see figure 5.)

In line B, 2...Rc7 is the correct reply to 2.Bd2. So we try this move in line A after 3. Bd2. White can then play bc3 mate in line A so we try Bc3 in line B, to which we discover Rxc3 is the refutation. Finally Rxc3 is tried as a refutation to 4. Be3 mate in line A. It is found to be illegal because the Black Knight on c6 blocks the Rook's path. We have therefore ascertained that 2. Bf4 was necessary in order to force the Black Knight to c6 where it blocks the path of Black's Rook from c7 to c3.

FIGURE 4

Figure showing the two lines of play that indicate the reason for the move 1. Bg5 in the position shown in figure 3.

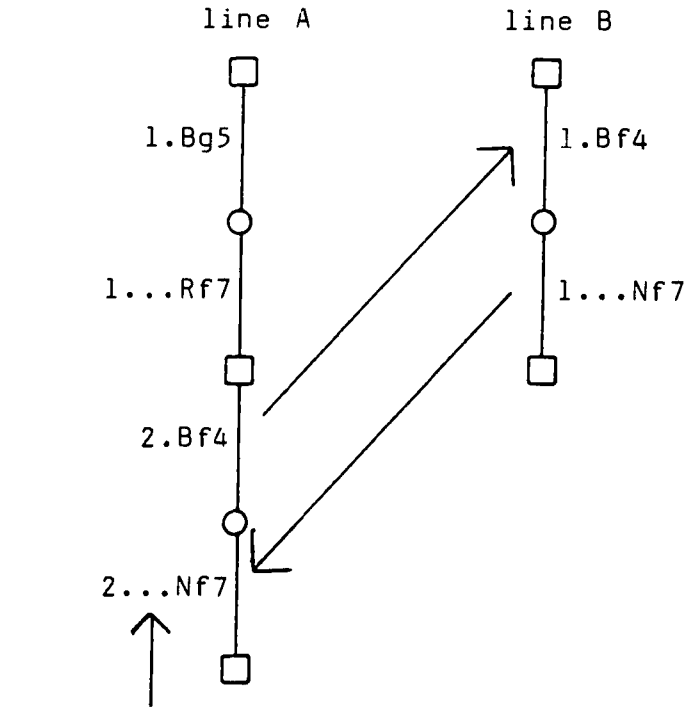
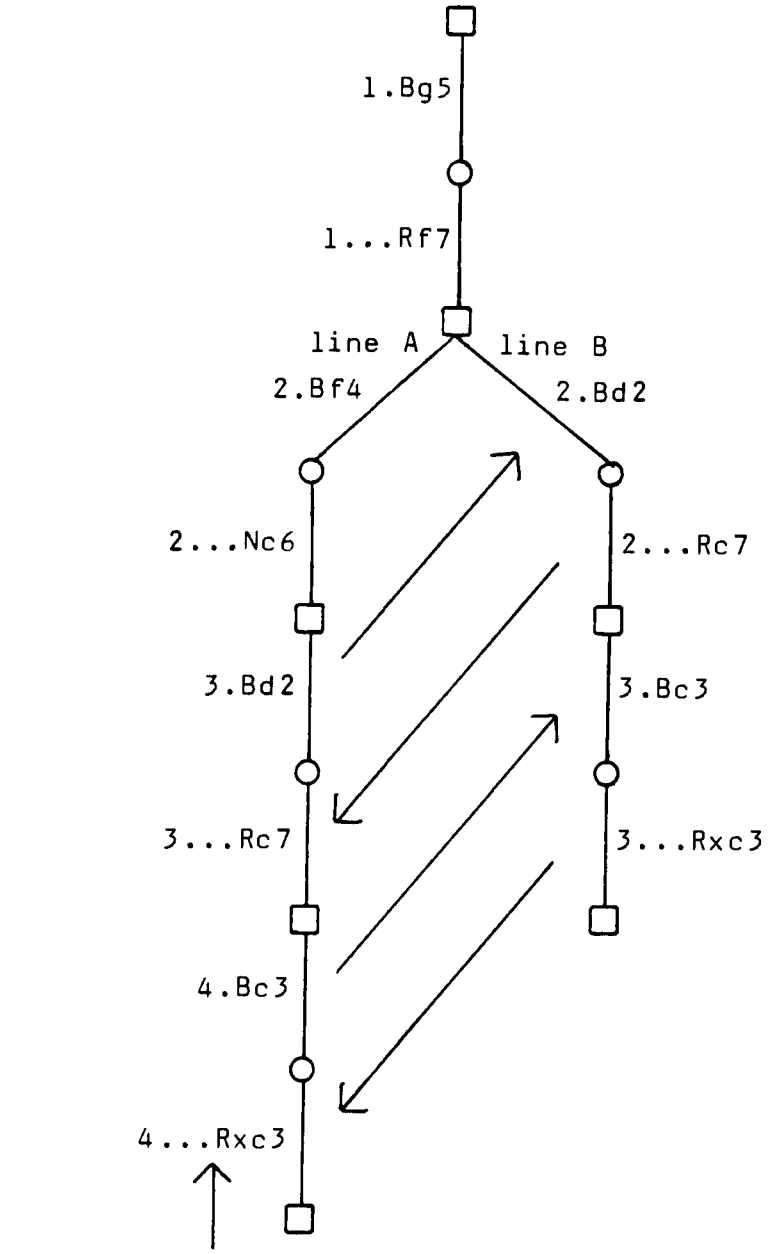


FIGURE 5

Figure showing the two lines of play that indicate the reason for the move 2. Bf4 in the position shown in figure 3.



4.3 The alternating search routine.

We can now define procedure ASR for performing the alternating search illustrated in the examples above.

Let P1, P2 be positions in a tree T. The procedure is called with two arguments A1 and A2. A1 is set to P1 and A2 is set to P2. The procedure can be called when the SM is at position P1 and Goal G. The flow chart in figure 6 outlines the procedure. L(1) and L(2) are lists of moves initially empty. m is a variable that stores a move (see figure 6).

Halt conditions

Let [P1,P2] be a move. The routine halts if any of conditions A, B or C below are satisfied.

A) An illegal move is tried on one of the lines which was legal on the other.

B) If [P1,P2] is a successful move found in one of the lines and [P1',P2'] is the corresponding move tried in the other line and

for P2 a UTM node

$$\text{tally}(P2) \geq G$$

$$\text{and } \text{tally}(P2') < G$$

or for P2 a TTM node

$$\text{tally}(P2) < G$$

$$\text{and } \text{tally}(P2') \geq G$$

C) A move tried by the routine is successful in both lines.

The above conditions are subsequently referred to as Halt conditions A, B and C. Let m be a move [P1,P2], P1' a position, PC the piece moved in m, SQ the destination square in m and G a goal value.
try(m,P1,G) tests

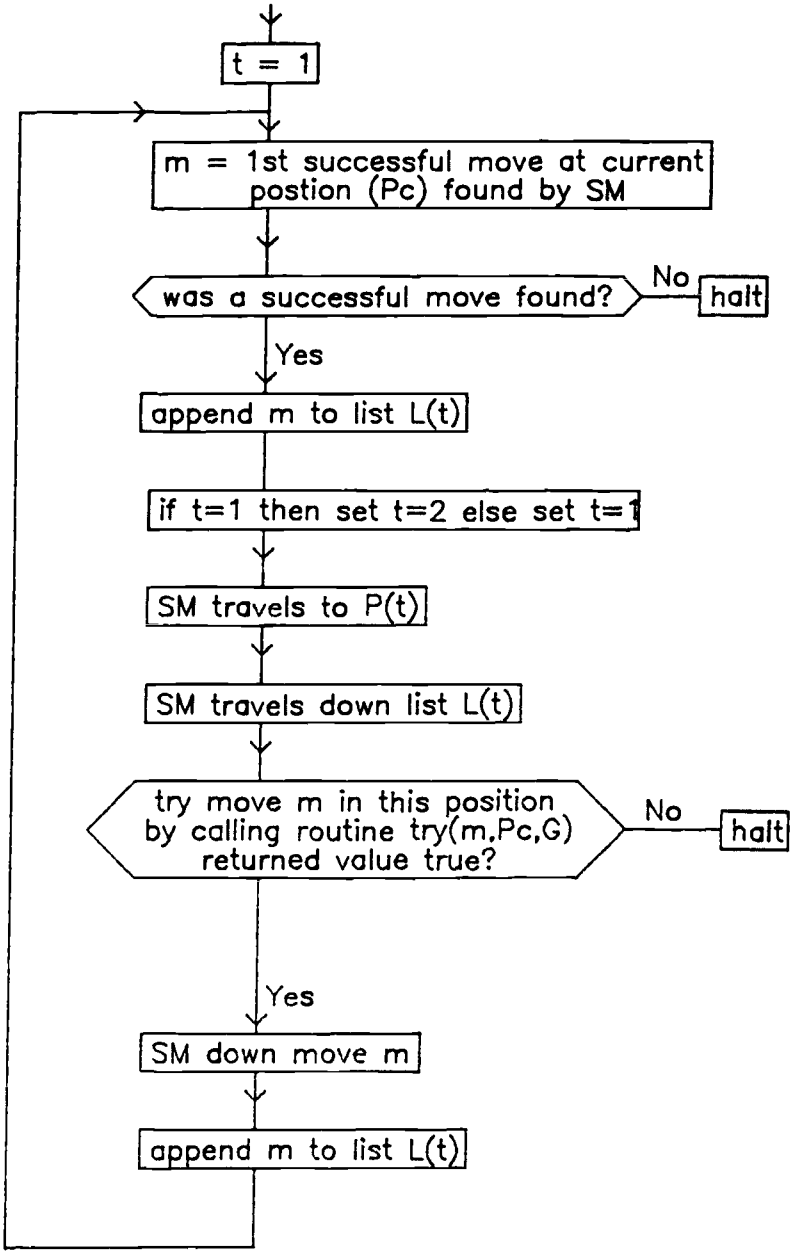
a) Whether moving PC to SQ is illegal in position P1.

b) If (a) is false and [P1',P2'] the move of PC to SQ then whether tally(P2) $\geq$ G and tally(P2') $<$ G
for P1 a UTM node.
or tally(P2) $<$ G and tally(P2') $\geq$ G
for P1 a TTM node.

The routine try(m,PC,G) returns FALSE if one of (a) or (b) above are true.

FIGURE 6

Flowchart for the alternating search routine.
try(m,Pc,G) is defined in the text.
Pc is the current position in the SM.
G is the goal value, m is a move variable
and L(1) and L(2) are move lists.



4.4 Producing a main variation.

Before examining causes for moves in a combination, it is useful to have a mechanism for extracting longest line variations from forcing trees searched by the machine.

Finding a longest line variation involves two stages:-

1) Finding a smallest depth D for which the machine finds the winning move.

2) Alternating between depths D and D-1 to find successive moves of the variation. This is exemplified below:-

Stage 1. finding the smallest depth.

1) The machine runs in its normal mode and returns the first move [P0,P1] of the combination. The goal G, iteration itn, and cycle number c, of the last search on which [P0,P1] was successful are stored.

2) The machine is set to terminate its search beyond a fixed depth D. D is even so that all nodes at depth D are UTM nodes. The machine acts as if all successor moves of nodes at depth D are illegal.

3) The SM returns to move [P0,P1] at cycle c, iteration itn, and goal G; and searches the tree below P1 again. If the move is successful, D is decreased by 2. Otherwise D is increased by 2. Step 3 is repeated until

two values of D; D1 and D2; are found s.t. D2 = D1 + 2 and [P0,P1] is successful when D = D2, and unsucccessful when D = D1.

Stage 2. Finding the main variation.

1) The SM is set to Goal G at iteration itn. D is set to D1. Since [P0,P1] was unsuccessful when D = D1 there must have been a move [P1,P2] that succeeded. The SM returns to node P1 and searches the tree below P1 to find the move [P1,P2].

2) D is set to D2. The SM returns to node P2 and searches the tree below P2 for a successful move [P2,P3].

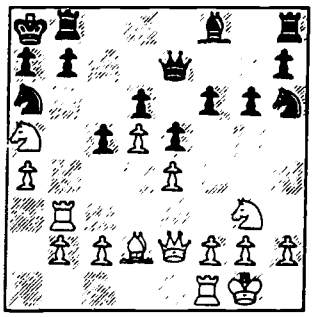
3) Steps 1 and 2 are repeated until a move [P(D1), P(D1+1)] is found. The sequence of moves [P0,P1], [P1,P2]...[P(D1),P(D1+1)] is then called a main variation.

Note that moves played by US in the main variation are successful and that moves played by THEM fail, but delay material loss maximally.

4.5 Multiple traces.

In position 13 VIRGO finds that White to play can achieve a gain of 3 material points. With the goal G = 3, the first five moves generated are:-

1. Qxa6 Pxa6
2. Rxb8ch Kxb8
3. Nc6ch Kc7
4. Nxe7 Bxe7
5. Bxh6



POSITION 13.

The move 2. Rxb8ch decoys the Black king to b8 where it is then a target of the Knight fork Nc6ch.
Let P1 be the position after 2...Kxb8 in the main line.
Let P2 be the position after 1...Pxa6 in the main line.
If the ASR is called between P1 and P2 then the trace needed to detect the decoy of the B. King is TRACE A, shown in figure 7.

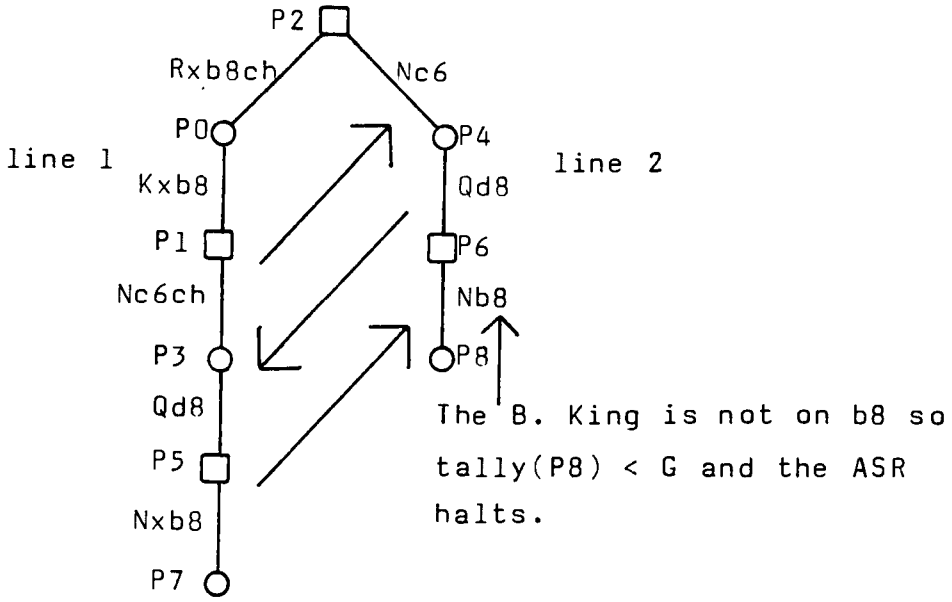
Both the moves [P4,P6'] = Rxb3 and [P4,P6] = Qd8 are successful moves. As the SM finds Rxb3 before Qd8 the

actual trace obtained is TRACE B shown in figure 7. It is thus necessary in general to make the ASR produce more than trace.

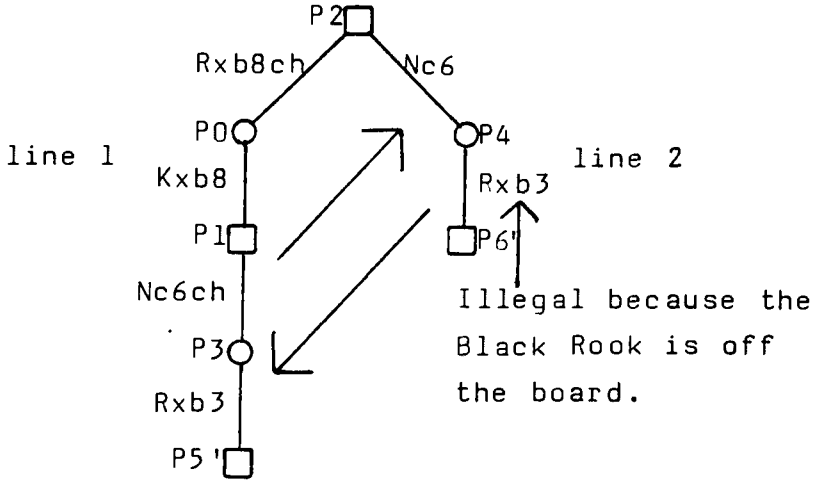
The ASR was modified so that at nodes where the SM is set to find a successful move, more than one successful move may be returned. A predefined limit is set on the number of successful moves sought at each node in order to limit the time taken to complete ASR calls.

FIGURE 7

Figure showing two ASR traces. Trace A brings to light a decoy of the B. King to b8. Trace B does not. P2 is position 13 after 1. Qxa6 2. Pxa6. The goal, G = 3.



TRACE A



TRACE B

4.6 Explaining a main variation.

There are four stages to explaining the moves in main variations generated by the VIRGO system.

Stage 1.

Generate a main line as described earlier and run the ASR between selected pairs of nodes related to the main line. Obtain "traces" from the ASR calls. A trace consists of the two move lists; L(1) and L(2) that are generated and the moves belonging to the shortest path between the two nodes initially presented to the ASR.

Stage 2.

Match the traces obtained in Stage 1 to templates, each template corresponding to a tactical motif such as "pin", "decoy" etc.

Stage 3.

For each template that matched in Stage 2 produce a description in English text.

Notation.

If L is a list of moves and n is a positive integer then L[n] represents the nth component of L. Whenever it is stated that the ASR is called between two nodes one of which has suffix 1 and the other suffix 2, it is implied that the ASR commences at the former. e.g. "The

ASR is called between P1' and P2'" implies the ASR starts at P1'.

4.6.1 Stage 1. Causal tests on a main line.

Each main line consists of a sequence of moves {[Pi, P(i+1)] | i=1,2,...n}. For each i for which $0 \leq i \leq n-2$, and for which i is even (so Pi is an UTM node):-

1) The ASR is called between A1 and A2 where

A1 = P(i+2)

A2 = Pi

(see figure 8.1). The traces returned are then applied to the templates for overload, decoy and linedecoy.

2) If cycle([Pi, Pi+1]) > 1 and [Pi,Pi+1] is not immediately goal achieving (i.e. tally(P(i+1)) < G) the ASR is called with:-

a) (i) A1 = N0

A2 = Nj

(ii) A1 = Nj

A2 = N0

for all Nj. N0 is s.t. [P(i+1), N0] is a dummy move and Nj is a successor position of P(i+1). (see figures 8.2(a & b). Each pair of traces returned is then matched against the templates: pinning, skewer and

fork.

b) Let m be a successful move found by the ASR at node N0 in (a) above. Move m will then be a threat contained in move [Pi,Pi+1]. When move m is tried by the ASR at node Nj, if it is illegal or unsuccessful at Nj, we can call [P(i+1),Nj] a defence to the threat. This will be the case iff the ASR does not halt because of Halt Condition C (section 4.3). If m is found to be a defence, a second trace is generated by calling the ASR with A1 = Nj, A2 = N0. (see figure 8.2(b)). The second trace is then matched against the pin template.

c) A1 = N0

A2 = P(i+2)

Where N0 is s.t. [P(i+1), N0] is a dummy move (see figure 8.2c). The traces returned are matched against templates guard, capthpc and moveaway.

d) A1 = N0

A2 = Pi

Where N0 is s.t. [P(i+1), N0] is a dummy move (see figure 8.2d). The traces returned are matched against the discovery template.

Definitions.

For each of cases 1, 2a, b, c and d above, let the MBE (move being explained) be the move in the main variation with

which any text produced as a result of a template match will be associated. If MBE is a White move, define REPLY to be the next Black move in the main variation. The MBE and REPLY moves are indicated for each case where appropiate in figure 8. For cases 2a and b, define a DEFMOVE to be [P(i+1),Nj] for each j. There are as many DEFMOVE's as there are successor moves of P(i+1).

FIGURE 8

Figure showing nodes at which different types of ASR calls are made. Pi is an UTM node in a main variation. A1 and A2 are the nodes initially presented to the ASR in each case. Moves MBE and REPLY and nodes NO and Nj are defined in the text.

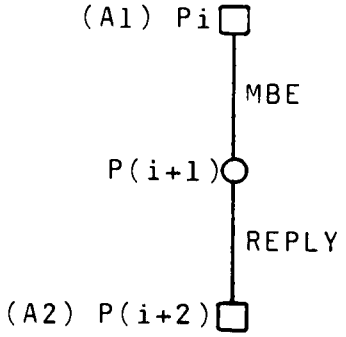


Fig 8.1

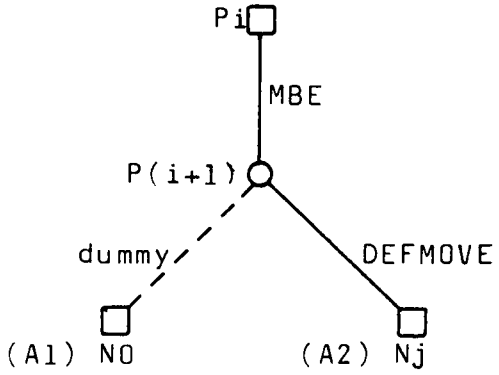


Fig 8.2a

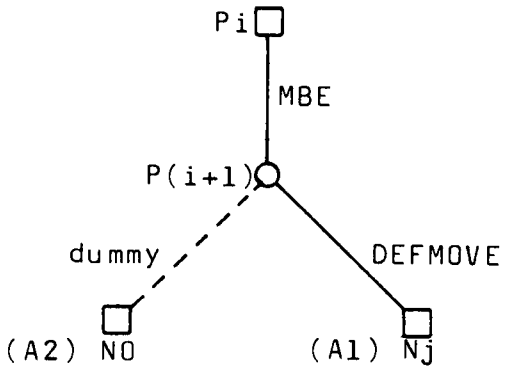


Fig 8.2b

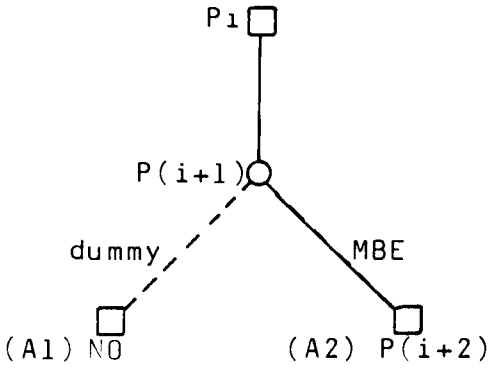


Fig 8.2c

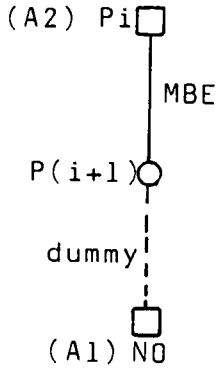


Fig 8.2d

4.6.2 Stage 2. Formalism for template attributes.

Each template comprises a set of logical conditions. When a trace is applied to a template, these conditions must be satisfied for the template to fire. The logical conditions are built up from lower level attributes of the applied trace. The following formalism sets out the attributes necessary to define these conditions.

Formalism for defining the templates.

If P is a position and PC a piece then

SQUARE(P,PC) = The square occupied by piece PC in position P.

OFFBOARD(P,PC) = TRUE if PC does not occupy any square in P.

If m is a move [P1,P2] say, then

PIECEMOVED(m) is the piece moved in m

ISCAPTURE(m) is TRUE if m is a capture and FALSE otherwise.

TAKENPIECE(m) is the piece capured on move m, if any.

TOSQUARE(m) = SQUARE(P2, MOVEDPIECE(m))

FROMSQUARE(m) = SQUARE(P1,MOVEDPIECE(m))

POSBEF(m) = P1

POSAFT(m) = P2

BLOCKED(P,PC,SQ) = TRUE iff PC could legally move to SQ in position P were it not for pieces blocking the path of PC to SQ. Can only be true if PC is a Bishop, Rook, Queen or Pawn on the 2nd rank.

BLOCKEDPIECE(P,PC,SQ) is the first blocking piece encountered when moving PC to SQ. Only applicable when BLOCKED(P,PC,SQ) = TRUE

POINTS(m) = The material gained for White by a capture or Pawn promotion on move m. POINTS(m) is negative (or zero) if m is a Black move.

VALUE(PC) = The material value of piece PC. This is 1 for a Pawn, 3 for a Knight or Bishop, 5 for a Rook, 9 for a Queen and 100 for a King. VALUE(PC) is positive for White and Black pieces.

COLOUR(PC) = The colour of piece PC.

WRONGSQUARE(P,PC,SQ) = TRUE if piece PC cannot legally move to SQ in position P except when the cause of the illegality is a piece or pieces blocking PC's path.

LINE1(TRACE) = List of moves L(1) generated by the ASR.

LINE2(TRACE) = List of moves L(2) generated by the ASR.

G(TRACE) = The goal setting during the ASR call.

CMS(TRACE) is the critical move that causes the ASR to halt. It is the last move in the line that terminates with a successful move.

THREAT(TRACE) is the move following the dummy move in TRACE. Only applicable to cases 2a, b, c and d of section 4.6.1.

Ps(TRACE) = POSBEF(CMS(TRACE))

Pf(TRACE) is the node at which trying move CMS(TRACE) caused the ASR to halt.

BOTHSIDESGOOD(TRACE) = TRUE iff the ASR halts because halt condition C is satisfied i.e. A successful move from one line tried at the corresponding node in the other is successful there too.

SCORECAUSE(TRACE) = TRUE iff the ASR halted on halt condition B. i.e. there is a legal move [Pf,Pf'] such that
MOVEDPIECE(CMS(TRACE)) = MOVEDPIECE([Pf,Pf']) AND
TOSQUARE(CMS(TRACE)) = TOSQUARE([Pf,Pf']) AND
for Pf a UTM node
tally(POSAFT(CMS(TRACE)) $\geq$ G(TRACE) AND
tally(Pf') $<$ G(TRACE)
or for Pf a TTM node
tally(POSAFT(CMS(TRACE)) $<$ G(TRACE) AND
tally(Pf') $\geq$ G(TRACE)

FORCED(TRACE) = TRUE iff
COLOUR(PIECEMOVED(CMS(TRACE))) = BLACK
and for each TTM node P in SLINE there is one and only one successful move [P,P+], where SLINE is the line containing the CMS in TRACE.

GOALSWING(m,TRACE) = TRUE iff

for COLOUR(m) = WHITE

tally(POSBEF(m)) < G(TRACE)

and tally(POSAFT(m)) ≥ G(TRACE)

or for COLOUR(m) = BLACK

tally(POSBEF(m) ≥ G(TRACE)

and tally(POSAFT(m) < G(TRACE)

i.e. move m swings the material tally above the goal for a White move, or below the goal for a Black move.

TEMPLATETYPE(m) is the name of the template that fired where m is an MBE from an ASR call of type 2c from the previous section. (i.e. CAPTHPC or MOVEAWAY or GUARD) or NULL if none fired.

Notes

1) Throughout the following description of templates, the TRACE argument is omitted from the trace attributes defined above whenever a template only requires one trace.

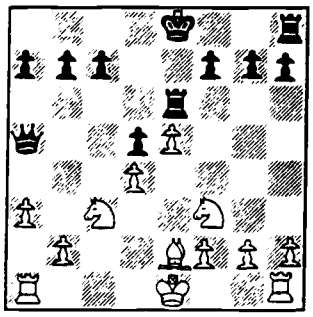
2) The condition BOTHSIDESGOOD(TRACE) = FALSE is a necessary condition for all templates unless otherwise stated.

4.6.3 The templates.

The decoy template.

A decoy is a move that induces an opponent's piece to a square where it later becomes a target for attack.
Two typical reasons for the opponent's piece moving onto the vulnerable square are to recapture material lost on the previous move, or to answer a threat by capturing the attacking piece. The decoy template defined below only fires when one of these two reasons apply.

In position 4 shown below, VIRGO finds that White to play can achieve a gain



POSITION 4.

of 7 material points. With the goal G = 7 the first five moves of the main line generated are:-

1. Pb4 Qb6
2. Nxd5 Qc6
3. Bb5 Qxb5
4. Nxc7 Kd7
5. Nxb5

The move 3. Bb5 decoys the Black Queen to b5 where it can later be taken by the White Knight (5. Nxb5).

Let P1 be the node before move 4. Nxc7ch in the main line.
Let P2 be the node before move 3. Bb5 in the main line.

When the ASR is applied between nodes P1 and P2 one of the traces returned is that shown in figure 9. The ASR halts at node P6 because tally(P7) = 8 $>$ G and tally(P8) = -1 $<$ G, so that halt condition B fires.

The conditions for the decoy template to fire are:-

1) (GOALSWING(MBE) AND GOALSWING(REPLY))
OR (TEMPLATETYPE(REPLY) = CAPTHPC)

AND 2) COLOUR(PIECEMOVED(CMS)) = WHITE

AND 3) PIECETAKEN(CMS) = PIECEMOVED(REPLY)

AND 4) SCORECAUSE

AND 5) tally(Pf) + VALUE(PIECEMOVED(REPLY)) $\geq$ G

In condition 1 the first clause of the disjunct implies that the MBE must swing the material tally above the goal G and

that the REPLY must swing the material tally below (or equal to) G. The second clause of the disjunct is satisfied iff the MBE contains a threat, and the REPLY defends against it by capturing the threatening piece.

In our example:-

1) GOALSWING(MBE)

= GOALSWING([P2,P0])

= FALSE

But

TEMPLATETYPE(REPLY)

= TEMPLATETYPE([P0,P1])

= CAPTHPC

Because [P0,P1](Qxb5) captures the threatening Bishop.

So condition 1 is satisfied.

Conditions 2 and 3 express the requirement that the CMS must be a move of a White piece and the piece moved in the REPLY (the decoyed piece) must be the piece captured in the CMS. In our example the piece moved in the REPLY ([P0,P1]) is the B. Queen which is the piece taken by the W. Knight in the CMS([P5,P7]). Condition 4 implies that

tally(Pf) + points(CMF) < G

where CMF is the move of PIECEMOVED(CMS)
to TOSQUARE(CMS) played in position Pf.

Condition 5 implies that a cause of condition 4 being satisfied is that PIECEMOVED(REPLY) was not captured during CMF. This in turn can only have been caused by REPLY. In our example, condition 4 holds because

tally(P7) = 8 > G
AND tally(P8) = -1 < G

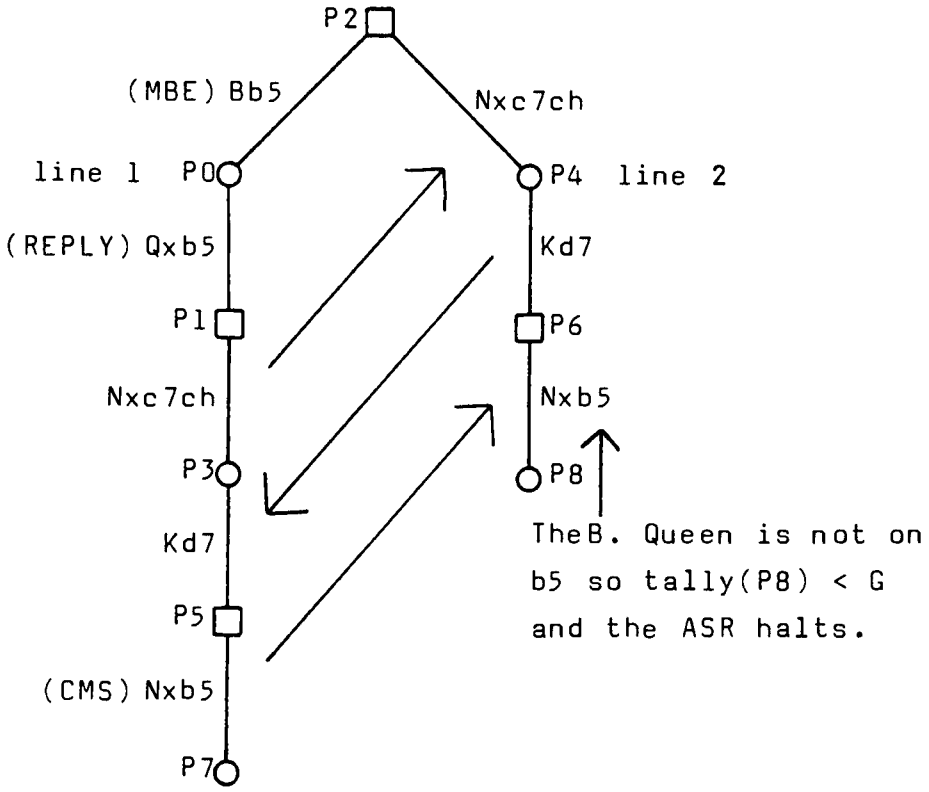
Condition 5 holds because

tally(Pf) + VALUE(PIECEMOVED(REPLY))
= -1 + VALUE(B. Queen)
= -1 + 9
> G

Conditions 1 to 5 are satisfied so the decoy template fires.

FIGURE 9

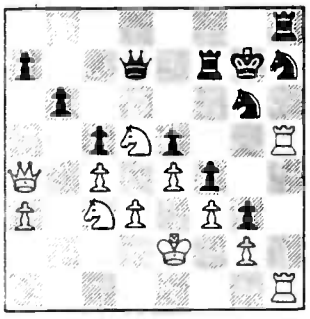
Figure showing an ASR trace that fires the decoy template. P2 is position 4 after 1. Pb4 Qb6 2. Nxd5 Qc6. The goal, G = 7.



The overload template

A piece is overloaded when it has to perform more than one defensive duty. Then in realising one defence it may be forced to neglect another with resulting loss of material. Another way of looking at an overload combination is as a type of decoy. The overloaded piece is decoyed to a square where it cannot fulfil some necessary defensive function.

In position 12 shown below, VIRGO system finds that White to play can



POSITION 12.

achieve a gain of 5 material points. With the goal G = 5, the first three moves of the main line generated are:-

1. Rxh7 Rxh7
2. Rxh7 Kxh7
3. Nf6ch Rxf6

The Black Rook (f7) is overloaded: it cannot guard against the Knight fork (3. Nf6ch) and protect its Queen from capture by the White Queen (e4).